COTENTS AT A GLANCE

Table of Contents

COTENTS AT A GLANCE....2
Foreword....4
About the Author....5
Preface....6
Chapter 1 Introduction....8
1.1 Origin of Processing....10
1.2 Setting up Processing....11
1.3 Launching the application....13
1.4 How to use this book....17
Chapter 2: Code Art....19
2.1 Code Artist....21
2.1.2 Ben Laposky, 1914–2000....21
2.1.3 John Whitney Sr., 1918–1995....21
2.1.4 Herbert W. Franke, b.1927....22
2.1.5 Lillian Schwartz, b. 1927....22
2.1.6 Harold Cohen, b. 1928....23
2.1.7 Roman Verostko, b. 1929....24
2.1.8 George Legrady, b. 1950....25
2.1.9 Casey Reas, b. 1970....25
Chapter 3: Theory of Recreational Coding....28
3.1 How does Processing works?....30
Chapter 4: Computer Graphics....40
4.1 Introduction....42

4.2 History of Computer Graphics 43
4.2.1 Output Technology....43
4.2.2 Input Technology....44
4.2.3 Graphics Formats....44
4.2.3.1 Raster graphics....45
4.2.3.2 Vector graphics....45
4.2.4 Animations....45
4.3 Digital Creativity....46
4.3.1 Creation of Creative Graphics 48
4.3.2 Storage of Creative Graphics....52
4.3.3 Manipulation of Creative Graphics....53
Chapter 5 Digital Creatives and Graphics Programming....60
5.1 Introduction 61
APPENDIX A 63
CreativePlanet Source Code 63
CreativeExample Source Code 63

Foreword

"**Digital Creatives**" is a skill building or developing topic in computer graphics based on the processing language. It will enable the skill developer to acquire amazing skills in art or code art using the amazing language. Digital creatives is a digital exercise of processing facility as a main source of pleasure and beauty. It is an account of recreational beauty and fascination available from the digital outputs of processing language. These creatives are digital proves of recreational coding of code art and not a formal proof of digital creatives. Digital creatives is an appreciation and enjoyment of digital (in)dependence of the results of processing facility or any computer graphics facility (OpenGL).

About the Author

Prof. Dr. Frank Kwabena Appiah holds Engineering Doctorate (E.D.) in computer and software engineering from Ghana Academy of Engineering, which he established in 2012, Msc in Advanced Software Engineering from University of London, King's College London in 2009 and Bsc in computer engineering from Kwame Nkrumah University of Science and Technology in 2008. Prof. Dr. F. K. Appiah holds numerous doctorates in applied computer science and mathematics from King's College London, King's College Cambridge, Imperial College London and more. Frank Kwabena Appiah was born in Takoradi, son of Yaw Appiah, in the Western region of Ghana in 1985 but Asante and Twi/English speaking-person. He did his primary and secondary and (under)graduate mainly in three regions of Ghana- Northern, Greater Accra and Ashanti and graduate studies in England. He is appointed in mind and during his studies was wondering about things of art life. The things of life here is about code art developed from computer graphics.

Preface

The aim is to show something of beauty and fascination as well as the enormous extent of shapes formed from the topics of vertex coordination in computer graphics, only trigonometric in mathematics (mainly cosine, sine calculation – used together or double cosine or double sine after each vertex application) and the graphic input (mouse to map the desired number of points for the repeating of vertex coordination. After a convinced a graphic shape, the author selects and presents a fixed number of points to the shape processing in the *Processing Environment*. However, it is hope that you will enjoy the shapes presented in this book dubbed "Digital Creatives" - A skill builder. This fact, coupled with the belief that this material can provide great enjoyment to the reader or developer, led me write this skill topic. The unfamiliar shapes will keep you going on practicing code art. The persistence on your part will be just typing the codes of the art as presented in this book, following the chain of reasoning on the snippets of code and this is the interesting part on the book.

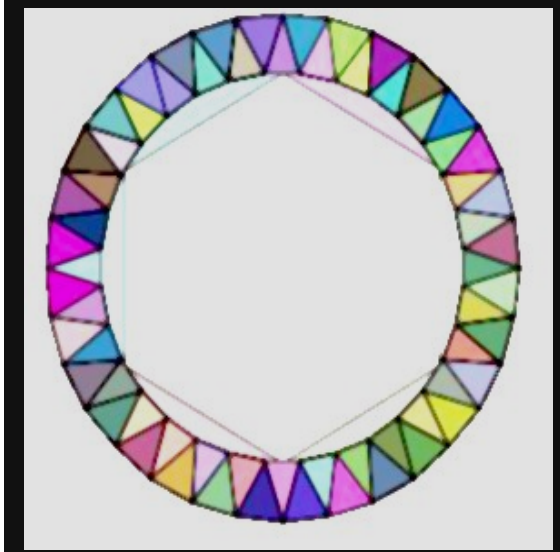
Department of Computer and Software,

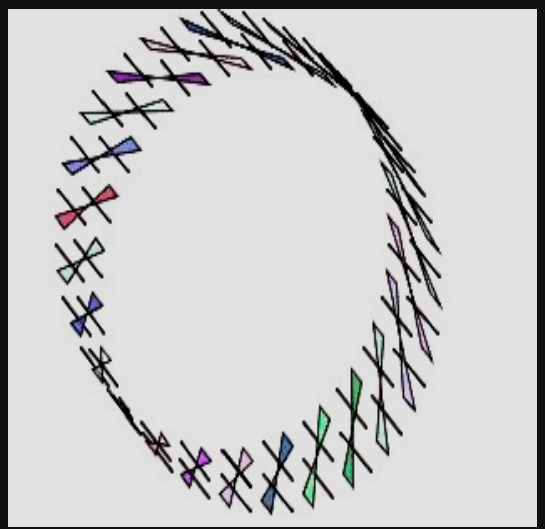
Ghana Academy of Engineering,

Ghana.

Chapter 1 Introduction

The computer is a computing machine and an instrument involved in the process of computer art history.





1.1 Origin of Processing

This book will introduce the origin of processing as done by the prior authors. Since around 1945 (or 1834 if you go back to Ada Lovelace's letter to Charles Babbage), many, many programming languages have been developed—in fact, more, though than most people would guess. Unless you are deep in the computer science world, you maybe have heard of at most five to ten different programming languages. According to a somewhat comprehensive language list at http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm, more than 2,500 languages have been developed. The guess is that the number is actually much higher. Interestingly, at the time of this writing, Processing isn't on the list but a phenomenal success. This might be due to the list not being properly maintained or because some purists might not see Processing as a separate language. *What*? Well, rest assured about Processing. But there is a valid argument to be made that Processing is not technically a language, but more of a programming environment— what is commonly referred to as an IDE, or integrated development environment. However, before you get into more Processing nuts and bolts, a more conceptual over-the-hood discussion about Processing and its forebear, DBN, will be helpful, or at least mildly interesting. Processing grew out of research in the Aesthetics + Computation group at the MIT Media Lab in around 2001. Prior to that, John Maeda, the group's famous leader, with the aid of some of his students, developed DBN. DBN was a very simple programming language and environment, begun in around 1999. Casey Reas and Ben Fry also participated in the development of DBN. Processing can be thought of as the very advanced, production-grade child of DBN. DBN was usable on the Web or as a stand-alone application, and was designed to massively simplify the coding process for visual artists.

Unlike Processing, DBN didn't have much application outside of the classroom, as the screen output was limited to about 1.5 by 1.5 inches and there was no color. Yet many of the basic processes we expect in a programming language were there, including the ability to set and recall values, create customized commands, and run loops. It even had some basic networking and event behavior (mouse and keyboard detection). If you're interested to learn more, as of this writing, the DBN site can still be found at: http://dbn.media.mit.edu/. DBN was developed to simplify the massive complexity involved in graphics programming.

Later on in the book, I will provide some sample code illustrating the code needed to generate the simplest graphics, even in a high-level language like Java. By "high level," I mean a language that is supposed to be easier to work in than a low-level language, assembly (machine) language. You can think of the lowest-level language as zeros and ones. DBN was in a sense a reaction to this problem, and it went, some would argue, overboard in addressing it. Processing, having the benefit of standing on the shoulders of DBN, as well as being a second go-around for Ben and Casey, cleverly addressed this problem by providing a multi-modal approach, allowing some users to work in a very high-level way—like in DBN— while other code heads

(myself included) could work in pure Java and still others could work with a hybrid approach. This totally inspired design, providing *choice*, believe what makes Processing a very special and cutting-edge tool/environment/language. Today, Processing, free from the confines of the Media Lab, is supported by Ben and Casey, with a growing cadre of devoted (some might say obsessed) "Processors" around the world.

1.2 Setting up Processing

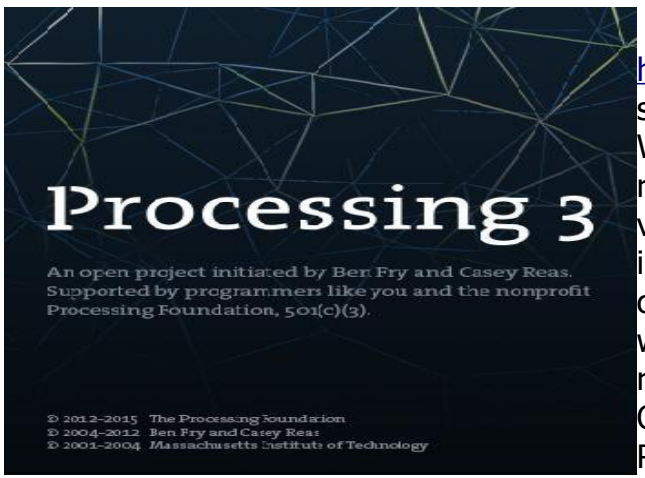
If you haven't already downloaded Processing, you should do so now. You'll (obviously) need a working copy of the software/language to follow the tutorials throughout the book. To download the latest version, go to http://processing.org/download/index.html. If you're not sure which version to download, keep reading. Since Processing is a Java application, any platform that can run Java can theoretically run Processing. However, Processing is only officially released for Windows, Mac OS X, and Linux, and the software is only extensively tested on Windows and OS X. Linux users aresomewhat on their own. Here's what the

> *For the Linux version, you guys can support yourselves. If you're enough of a hacker weenie to get a Linux box set up, you ought to know what's going on. For lack of time, we won't be testing extensively under Linux, but would be really happy to hear about any bugs or issues you might run into . . . so we can fix them. As of this writing, the latest down-loadable version of the software is 3.1.1, released July 29, 2016. It's possible, by the time you're reading this, that the release number has changed, as the developers are in the process of stabilizing the current beta release. Any changes made to the language between release and version should be very minor and primarily focused on debugging existing functionality. For more information about the different releases, check out* http://processing.org/download/revisions.txt.

Processing site says in regard to Linux users: For more details about platform support, please check out http://processing.org/reference/environment/platforms.html#supported. In selecting a version to download, Mac and Linux users have only one choice; Windows users have two choices: Processing with or without Java. The recommendation is to download Processing with Java. However, the without-Java version is available if download size is an issue and you know you have Java installed. If you're not sure whether you have Java installed, and/or the idea of changing your PATH variable gives you the willies, please download Processing with Java. If you still want to download them separately, here's a link (but

remember, you've been warned): http://java.sun.com/javase/downloads/index.jsp. OS X users already have Java installed, thanks to the good people at Apple. Regarding Java, the most current version available on Windows is Java SE 6 (the SE stands for Standard Edition). On OS X, Java releases typically lag behind, and the most current version is J2SE 5 (the names are also annoyingly a little different). The most current version on Linux is also J2SE 5. If all this isn't confusing enough, Processing only supports J2SE 1.4 Processing. However, Processing is only officially released for Windows, Mac OS X, and Linux, and the software is only extensively tested on Windows and OS X. Linux users are somewhat on their own. Here's what the Processing site says in regard to Linux users:

> *For the Linux version, you guys can support yourselves. If you're enough of a hacker weenie to get a Linux box set up, you ought to know what's going on. For lack of time, we won't be testing extensively under Linux, but would be really happy to hear about any bugs or issues you might run into . . . so we can fix them.*


Processing 3
An open project initiated by Ben Fry and Casey Reas.
Supported by programmers like you and the nonprofit
Processing Foundation, 501(c)(3).
© 2012–2015 The Processing Foundation
© 2004–2012 Ben Fry and Casey Reas
© 2001–2004 Massachusetts Institute of Technology
Figure 1: Processing Start-up.

For more details about platform support, please check out http://processing.org/reference/environment/platforms.html#supported. In selecting a version to download, Mac and Linux users have only one choice; Windows users have two choices: Processing with or without Java. The recommendation is to download Processing with Java. However, the without-Java version is available if download size is an issue and you know you have Java installed. If you're not sure whether you have Java installed, and/or the idea of changing your PATH variable gives you the willies, please download Processing with Java. If you still want to download them separately, here's a link (but remember, you've been warned): http://java.sun.com/javase/downloads/index.jsp. OS X users already have Java installed, thanks to the good people at Apple. Regarding Java, the most current version available on Windows is Java SE 6 (the SE stands for Standard Edition). On OS X, Java releases typically lag behind, and the most current version is J2SE 5 (the names are also annoyingly a little different). The most current version on Linux is also J2SE 5. If all this isn't confusing enough, Processing only supports J2SE 1.4 and earlier (yes, J2SE 5 and Java SE 6 come after J2SE 1.4). Version 1.4 is the version that comes bundled with Processing's Windows installer, and it is also the default installation in OS X. The reason Java version numbers go from 1.4 to 5 is because Sun, in their wisdom, decided to drop the "1." from the names—if you really care why, you can read about it here: http://java.sun.com/j2se/1.5.0/docs/relnotes/version-

5.0.html. What all this means to Processing users is that you can't use any new Java syntax specified in releases after 1.4 within the Processing development environment. (Syntax is essentially the grammar you use when you write code) For the latest information on between Processing and Java, please see http://processing.org/faq.html#java.

1.3 Launching the application

- **OS X**: After downloading the software, launch the Stuffit X archive (.sitx), which will create a Processing 3.1.1 folder. Within the folder you'll see the Processing program icon.
- **Windows**: After downloading the software, extract the ZIP archive (.zip), which will create a Processing 3.1.1 folder. Within the folder you'll see the Processing program icon.

To test that Processing is working, double-click the Processing program icon to launch the application. A window similar to the one shown in Figure 1 should open.

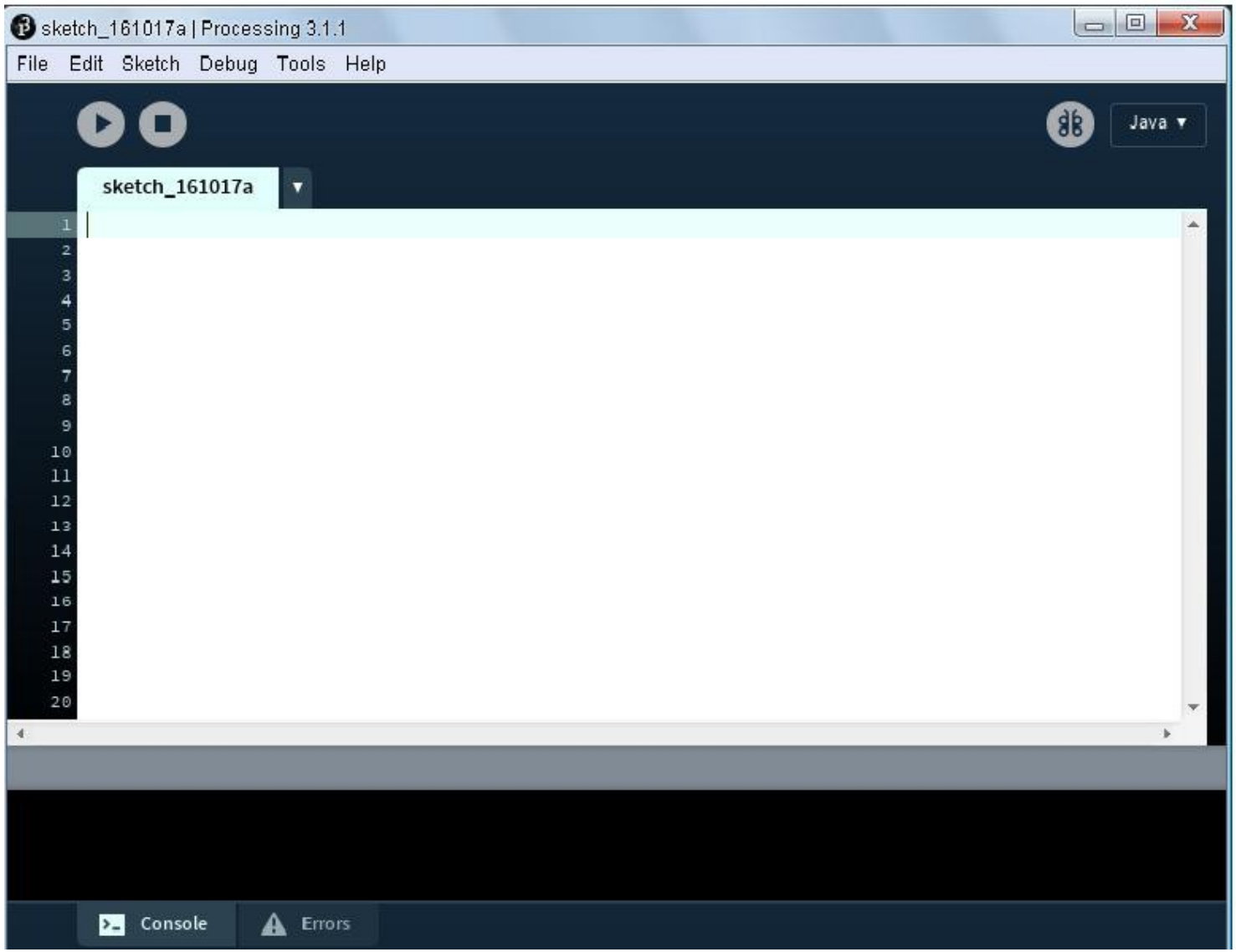


Figure 2 : Processing Application Interface.

Processing comes with a bunch of cool code examples. Next, let's load the Structure ► Coordinates example into the Processing application. You can access the example, and many others, through Processing's File menu, as follows: Select File ► Sketchbook ► Examples ► **Structure** ► **Coordinates** from the top menu bar. You should see a bunch of code fill the text-editor section of the Processing window, as shown in Figure 3.

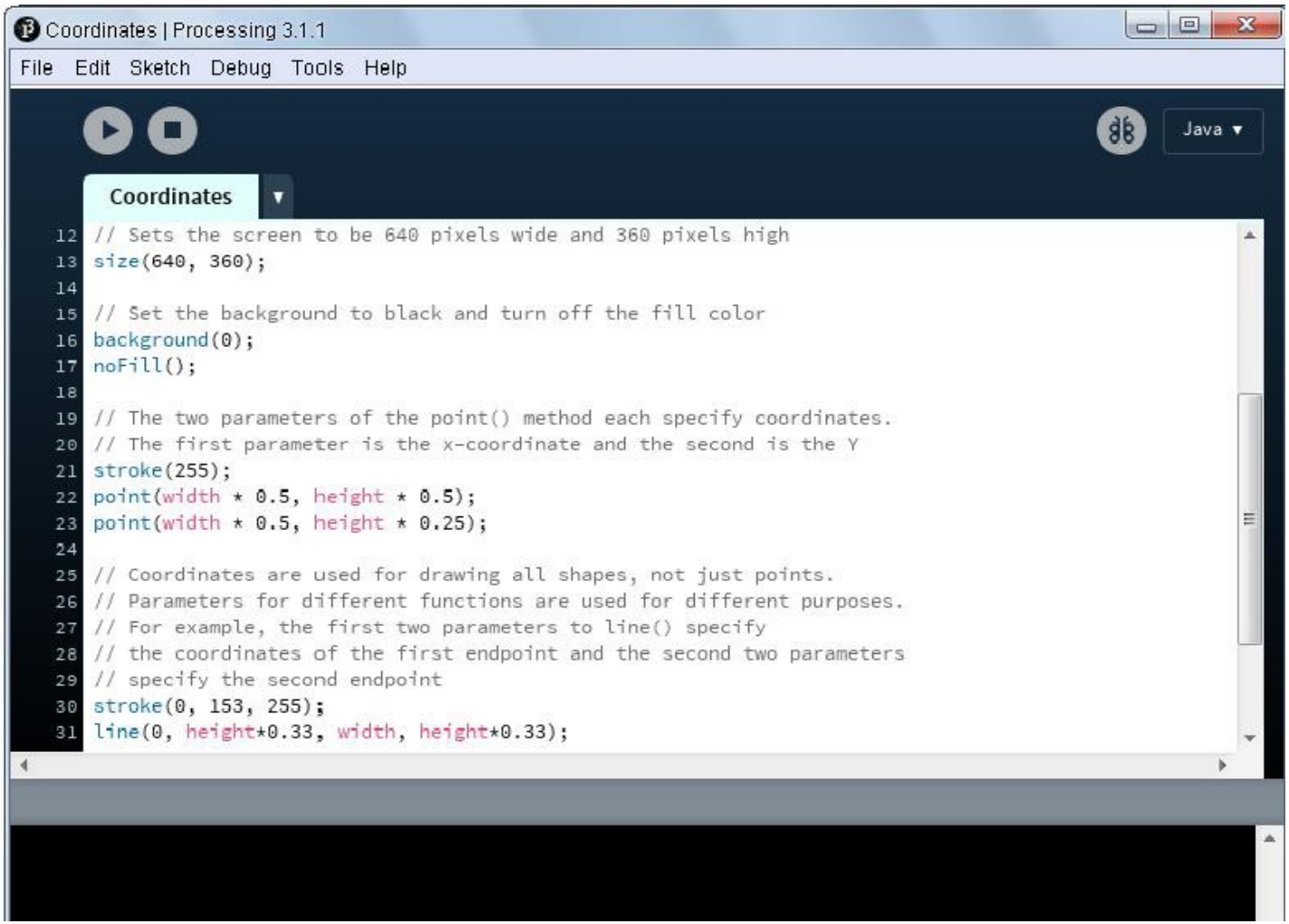

Coordinates | Processing 3.1.1
File Edit Sketch Debug Tools Help
Java
Coordinates
12 // Sets the screen to be 640 pixels wide and 360 pixels high
13 size(640, 360);
14
15 // Set the background to black and turn off the fill color
16 background(0);
17 noFill();
18
19 // The two parameters of the point() method each specify coordinates.
20 // The first parameter is the x-coordinate and the second is the Y
21 stroke(255);
22 point(width * 0.5, height * 0.5);
23 point(width * 0.5, height * 0.25);
24
25 // Coordinates are used for drawing all shapes, not just points.
26 // Parameters for different functions are used for different purposes.
27 // For example, the first two parameters to line() specify
28 // the coordinates of the first endpoint and the second two parameters
29 // specify the second endpoint
30 stroke(0, 153, 255);
31 line(0, height*0.33, width, height*0.33);

Figure 3 : Coordinates Application

To launch the sketch, click the right-facing run arrow (on the left of the brown toolbar at the top of the Processing window—it looks like a VCR play button), or press Cmd+R (OS X) or Ctrl+R (Windows). If you were successful, a 640-pixel-by-360 pixel display window with a dark background should have popped open, showing an orange rectangle with a blue line crossing it in the window (see Figure 4). Congratulations! You've just run your first Processing sketch. I recommend trying some of the other examples to get a taste of what Processing can do and to familiarize yourself a little with Processing's simple yet elegant interface.

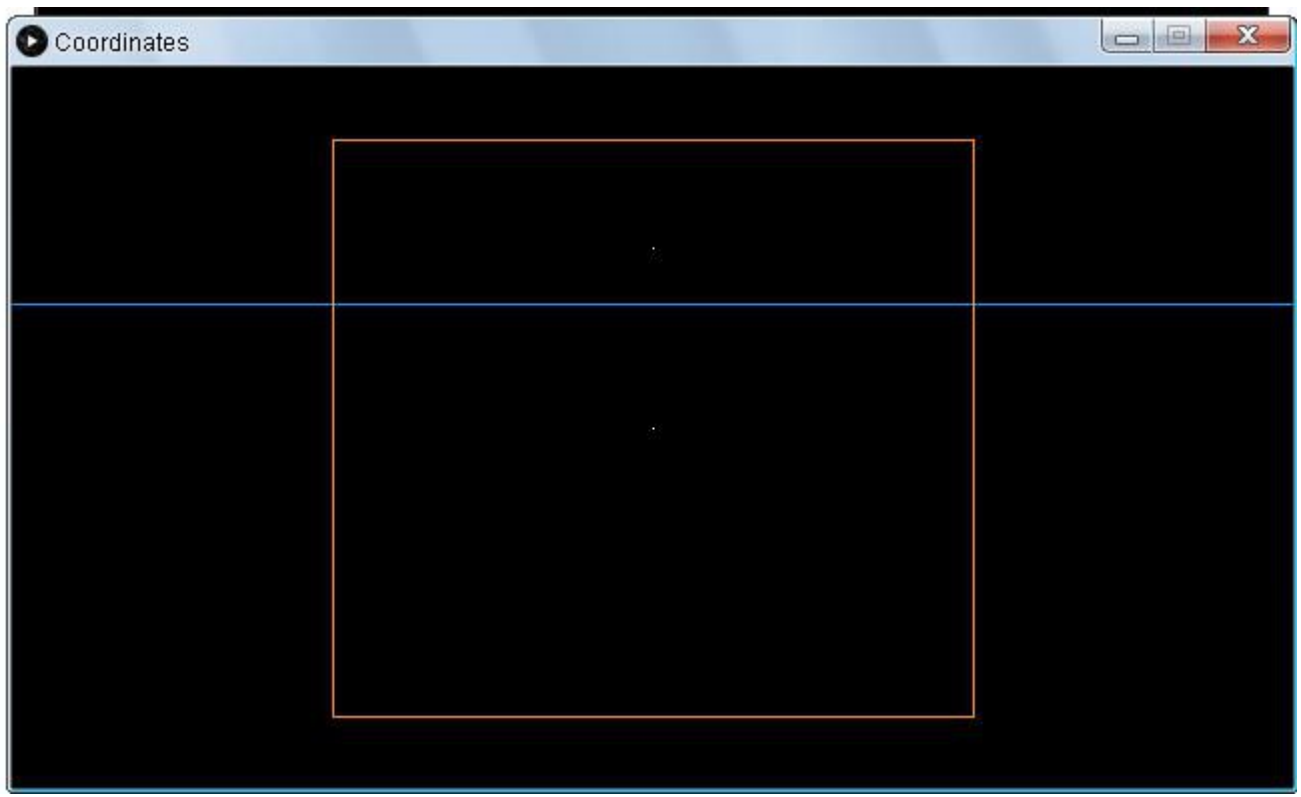


Figure 4: Coordinate Example Running

1.4 How to use this book

The main aims of the book is to

- Enable the skill developer to acquire amazing skills in art or code art using the amazing language.
- Experience digital exercising of processing facility as a main source of pleasure and beauty.
- Account of recreational beauty and fascination available from the digital outputs of processing language.
- Proof creatives as a digital proves of recreational coding of code art and not a formal proof of digital creatives.
- Appreciate and enjoy digital (in)dependence of the results of processing facility.

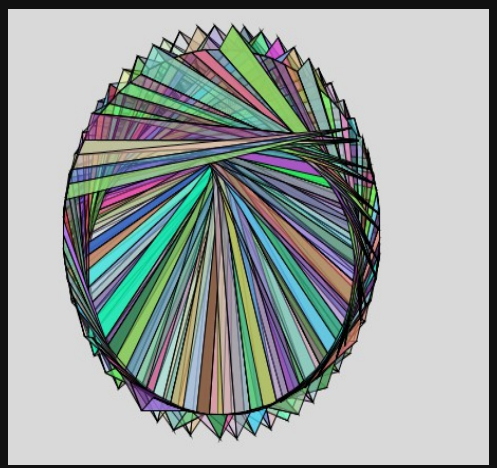
The objectives to introduce processing itself as a home creative medium including some history and theory of the subject topic or matter. Based on my own experience wanted to present these creatives in this book as a digital creatives founded mainly on only a code art and exploited in details. This text provides a friendly artist and colorful creative works and can be used as basic graphics programming. Finally, take on the excitement provided by processing application environment and learn graphics generally.

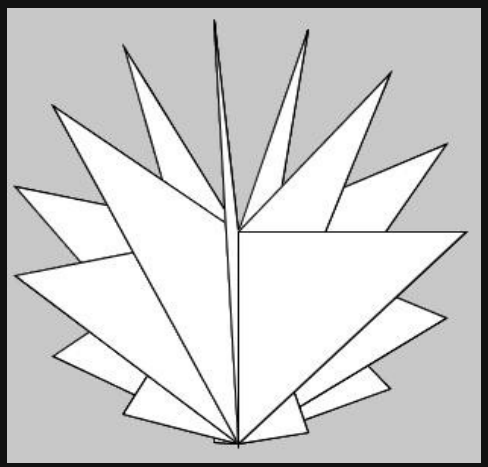
Learning the digital creatives will require:

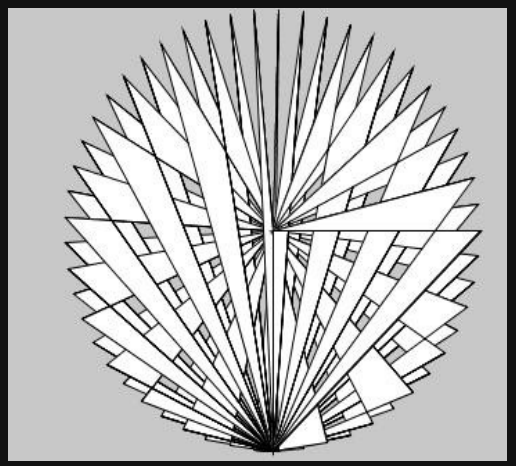
1. Little effort to memorize the basic snippet in Chapter 2.
2. No effort to break or edit the presented code.
3. Reading the code snippet labeled as shape over and over until it sticks.
4. Little effort to follow the chapters but not linearly. Read and practice about 5 illustrations per day.
5. A little programming skills in Java.

Chapter 2: Code Art

The creative process is a highly integrated activity reflecting history, aesthetic theory, and often the technological breakthroughs. This was certainly the case during the Renaissance, when artists, engineers, scientists, and thinkers all came together to create truly remarkable works of art and engineering.







2.1 Code Artist

The code artist of the computer art era did apply computation and technology to their creative process. This text informs the reader, few of the biographies of the listed artist in their facilitation to computer art creation and process. The computer is a computing machine and a few other people were involved in the process of computer art history namely: Charles Baggage, professor of Mathematics at Cambridge, Ada Lovelace, inventor of Ada Language, Konrad Zuse, developer of Z1, claimed as the first computer and other computer initiators like John Atanasoff's ABC computer, the Mark 1 at Harvard, and the ENIAC at Penn

2.1.2 Ben Laposky, 1914–2000

Both mathematician and artist, Ben Laposky was a pioneer among computer art pioneers. He created the first electronic abstract images in 1950 by using an electronic analog device called a cathode ray tube oscilloscope. To put Laposky's pioneering role in some context, it wasn't until 1960 (ten years later) that the term "computer graphics" was first coined by William Fetter, a graphic designer at Boeing. Laposky created images of beautiful mathematical curves he named "oscillons." The curves were based on the basic waveforms utilized in analog computers. Analog computers, now mostly unknown, were utilized as early as the 1920s and were capable of very fast, complex calculations. They used continuous variations in voltage, allowing real-time calculations, as opposed to distinct finite signals, which are employed in digital computers. To learn more about analog computers, check out http://en.wikipedia.org/wiki/Analog_computer. By the 1940s, analog computers began to be replaced by digital computers, which were more affordable to produce and more universal as general-purpose tools. I'll discuss more about the type of waveforms Laposky generated later in the book, including how to generate some of them. To capture the curves, Laposky photographed the images directly off the display with high-speed film. What I find striking about his images is their elegant simplicity and organic quality, in contrast to the technical means from which they arose. Beyond Laposky's pioneering technical virtuosity is his steadfast focus on the aesthetic quality of his work. This combined focus on the technical and aesthetic is an important enduring legacy that unites all the artists mentioned in this chapter. To learn more about Laposky and view some images of his work, check out www.dam.org/laposky/index.htm and www.atariarchives.org/artist/sec6.php.

2.1.3 John Whitney Sr., 1918–1995

John Whitney's work involved photography, film, and eventually also music. In the 1940s, he began to study images in motion, along with his brother James. This interest would remain with him the rest of his long career and eventually involve the

integration of motion and sound. Whitney combined a very successful career in commercial film and television with his more personal and experimental investigations. He even produced the title sequence for Hitchcock's *Vertigo*. By the 1960s, Whitney had formed a company, Motion Graphics, to produce commercials, at which he pioneered early computer-based animation and typography with a mechanical analog computer that he invented. Whitney gained wide recognition for his work with the analog computer, as well as for a sample reel entitled "Catalog," which showcased his innovative motion work. Between 1966 and 1969, Whitney got to work with powerful digital computers as the first artist in residency at IBM. From this period through the '70s, he continued to push the complexity of his work, creating numerous experimental films and documentaries while also refining his focus to achieve what he called "harmonic progression." Throughout his entire career, Whitney remained an innovator, eventually developing a computerized instrument that allowed him to compose visuals and music in real time. Additional information on Whitney, including images, can be viewed at www.siggraph.org/artdesign/profile/whitney/nowhitney.html and http://en.wikipedia.org/wiki/John_Whitney_(animator).

2.1.4 Herbert W. Franke, b.1927

Franke is a brilliant generalist, with formal training in physics, mathematics, chemistry, psychology, and philosophy. He is a widely published author—his works include seven science fiction titles, essays, articles, and scripts on subjects as wide ranging as physics, theoretical chemistry, scientific photography, visual perceptions, art and technology, futurology, and speleology (the scientific study of caves). His early work paralleled Loposky's, utilizing oscilloscopes to generate images based on waveforms. Since 1962, Franke has remained active in "experimental aesthetics," lecturing and creating new work in the field. He cofounded the Ars Electronica Festival in 1979, arguably the most prestigious international electronic arts festival in the world, held annually in Linz, Austria. In 2005, for their work on Processing, Casey and Ben won a Golden Nica—the festival's top prize. Franke has remained on the bleeding edge of emerging technology throughout his career, and beginning in the mid '90s began developing algorithmic animations using the Mathematica software system. In 2002, he released a book on the subject, entitled *Animation with Mathematica*. To learn more about Franke (in English), see http://en.wikipedia.org/wiki/Herbert_W._Franke and www.dam.org/franke/index.htm; to learn more about him in German, see www.zi.biologie.uni-muenchen.de/~franke/.

2.1.5 Lillian Schwartz, b. 1927

Lillian Schwartz has been a central and pioneering figure in the history of computer art, making broad and significant contributions in computer-generated art and computer aided art analysis. Her work and research have led to advances in

graphics, film, video, animation, special effects, virtual reality, and multimedia. She gained prominence in the late ’60s when one of her kinetic sculptures, *Proxima Centauri*, was included in the groundbreaking 1968 Machine Exhibition at the Museum of Modern Art (MoMA), New York. She is also the first artist to have a piece of computer art acquired by the MoMA. In addition to the MoMA, her work has been exhibited at and is owned by the Metropolitan Museum of Art, the Whitney Museum of American Art, the Moderna Museet (Stockholm), Centre Beaubourg (Paris), the Stedelijk Museum of Art (Amsterdam), and the Grand Palais Museum (Paris). Her films have been shown and won awards at the Venice Biennale, Zagreb, Cannes, and the National Academy of Television Arts and Sciences; and she has been nominated for and received Emmy awards. Like many of the early pioneers, Schwartz has made substantial contributions within industry, academia, and the art world. She was a consultant at AT&T Bell Laboratories, IBM’s Thomas J. Watson Research Laboratory, and Lucent Technologies Bell Labs Innovations. Her research has led to advances in the understanding of visual and color perception, sound, perspective, historic preservation, and applications of the computer in the philosophy of art—using computation to investigate the creative process. She has taught in both fine art and computer science departments throughout the United States, as well as internationally. Schwartz has received numerous awards, including Computerworld Smithsonian awards in three categories. She has also been the subject of articles, books, and television news and documentary programs, and is the coauthor of *The Computer Artist's Handbook* (together with Laurens R. Schwartz). To learn more about Schwartz and see images of her work, go to her personal website, www.lillian.com/.

2.1.6 Harold Cohen, b. 1928

Cohen, probably more than anyone else, made a wide and dramatic migration from the traditional arts to computer art. He was a celebrated painter in the ’60s, including representing Great Britain in the Venice Biennial in 1966. In 1968, he moved to Southern California as a visiting faculty at the University of California, San Diego. He soon became interested in artificial intelligence, and in 1971 was invited to spend two years at the Artificial Intelligence Laboratory of Stanford University as a guest scholar. Since then, his work has focused on the integration of artificial intelligence with the visual artist’s creative process. Over three decades, Cohen developed an expert drawing system, a rule-based software program capable of simulating aspects of human creativity and intelligence. He named the system he developed AARON. AARON is capable of creating original works of art on its own. Although all the work AARON produces is stylistically similar, the range of subject matter, composition, and color is not. What is so surprising about AARON, and I believe what makes it groundbreaking, is that the work it produces is representational. Most algorithmic systems generate abstract or color field–oriented images. AARON creates, in a sense, freehand drawn images of people, still lifes, and landscapes. Cohen’s (and AARON’s) work has been shown around the world, including in the Los Angeles County Museum, Documenta 6, the San Francisco MoMA, the Stedelijk Museum in Amsterdam,

the Brooklyn Museum, the Tate Gallery in London, and the IBM Gallery in New York. Cohen has a permanent exhibit in the Computer Museum in Boston, and represented the United States in the Japan World Fair in 1985. In 2000, artificial intelligence entrepreneur and inventor Ray Kurzweil licensed the code to AARON and created a screen saver version of the software, which he hosts at his Kurzweil CyberArt Technologies website (www.kurzweilcyberart.com/). Cohen continues to push his work and process, developing even more naturalistic and organic representational images based on machine intelligence algorithms. To learn more about Cohen and to see images of his (and AARON's) work, check out www.kurzweilcyberart.com/aaron/hi_cohenbio.html and www.viewingspace.com/genetics_culture/pages_genetics_culture/gc_w05/cohen_h.htm. Here's an interesting blog entry he wrote about "creative" programming: http://grandtextauto.gatech.edu/2003/06/17/harold-cohen-on-artist-programmers/.

2.1.7 Roman Verostko, b. 1929

Roman Verotsko has had a long, distinguished, multifaceted career. For 16 years, beginning in 1952, he lived as a Benedictine monk. Toward the end of this monastic period, he was sent to New York to develop himself as an artist, with the goal of bringing this experience and training back to the abbey. While in New York, he received an MFA from Pratt and then took graduate courses in Art History at both Columbia and New York University. Eventually struggling with his beliefs in revelation and the dogmas of the church, he left the monastic life in 1968 and took a faculty position at Minneapolis College of Art and Design, where he remained throughout his academic career. Verostko spent the first 30 years of his artistic life painting. This early work dealt with the search for an internal dynamic force in a work of art. He wrote the following about his process: ın 1970, he received a Bush Fellowship grant, which provided him the opportunity to work with Gyorgy Kepes at the Center for Advanced Visual Studies at MIT. At the time, Verostko's interest in computer technology related more to how artists could humanize the emerging medium than how it could be applied directly to the creative practice; it would be 14 years (1984) before Verostko began utilizing computer code within his own creative practice. Eventually, code would become his primary creative medium, extending (rather than supplanting) his earlier interests in spiritual practice and painting. Verostko was influenced by artists such as Mondrian, Kandinsky, and Malevich in his pursuit to uncover the underlying unity of nature, based on an integration of opposing forces within the image. He developed a software/hardware system he called "Hodos." Hodos generates both screen-based and physical images created with brushes mounted on a pen plotter's drawing arm. Verostko's algorithms control the range of expression of the system, allowing for his personal style and interest in opposing forces to emerge. In 1995, Verostko and a group of artists including Jean-Pierre Hébert and Ken Musgrave formed an informal group they named "the Algorists," for those artists who use their own algorithms for creating art objects. Verostko has received numerous awards and recognition for his work including the Golden Plotter (first prize, 1994), Gladbeck (Germany), and Prix

Ars Electronica (honorary mention, 1993). Verostko maintains an excellent site about his work and life at www.verostko.com/.

2.1.8 George Legrady, b. 1950

Chronologically, Legrady represents a generation of digital artists born between the earliest pioneers—Laposky, Whitney, Franke, and Schwartz, who depended upon (and often were) scientists and technical experts—and the current generation of digital artists in their 30s and 40s, who grew up with the ease and convenience of personal home computing. Paralleling this is Legrady's own personal story. He is a two-time immigrant, born in Budapest in 1950, who fled to Montreal in 1956 during the Hungarian uprising, and then eventually settled in California in 1981. This recurring theme of being between or crossing generations, space, cultures, and even time runs throughout his work. Trained originally as a classical musician, Legrady's musical interests eventually crossed over, and in 1969 he worked as a rock and roll keyboard player. A product of the counterculture revolution of the '60s, Legrady created socially conscious documentaries in the 1970s and '80s on the Cree Indians of northern Quebec, communist iconography of central Europe, and handpainted advertising billboards in China. In 1981, Legrady began working with digital artist pioneer Harold Cohen (whose bio is included in the preceding text), experimenting with computer code as a creative medium. By the mid-'80s, Legrady, then an assistant professor at the University of Southern California, began to receive recognition and awards for his digital work, which at the time consisted of digital prints. In the following decade and through the present, Legrady's work has become more site-specific and computationally intensive, involving interactive and algorithmically-based installations. For example, his well-known piece *Pockets Full of Memories* involves an interactive space with large-scale projections. Here's an excerpt about the piece, taken directly from Legrady's site (http://www.mat.ucsb.edu/~g.legrady/glWeb/Projects/pfom2/pfom2.html):

George Legrady is Professor of Interactive Media, with joint appointments in the Media Arts and Technology program and the department of Art at UC Santa Barbara. Additional information about the artist and his work can be found at www.georgelegrady.com/.

2.1.9 Casey Reas, b. 1970

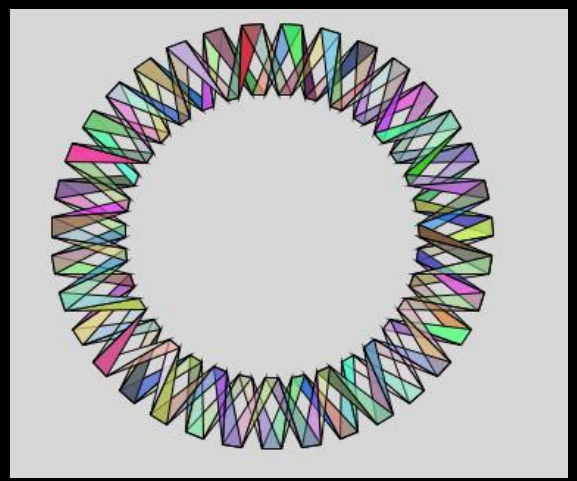
Casey Reas is a renowned designer/artist, lecturer, thinker, and of course one half of the Processing design/development team.

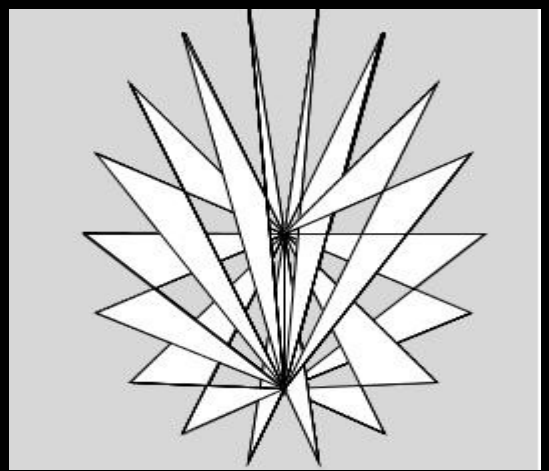
He was a student of John Maeda's at the MIT media lab, where he and Ben Fry helped develop the DBN programming language/environment. Processing was, to a degree, a natural outgrowth of DBN. Reas originally studied design at the University of Cincinnati before attending MIT, and was one of the founding professors at Interaction Design Institute Ivrea. At Ivrea, Reas worked with an international student body to develop a new arts pedagogy. It was during this period in Ivrea that he and Fry initiated Processing. Reas's work is software-based—code and coding are his primary medium. However, he manifests his code-based work in a variety of executions, including kinetic, reactive, and printed pieces. Some of his most recent work employs ideas explored in conceptual and minimal artworks, such as the wall drawings of Sol LeWitt. In a commission he received in 2004 from the Whitney, entitled *{Software} Structures*, Reas explored the relationship between conceptual art and software art. For the project, Reas created three unique descriptive structures that merely described relationships between elements in the pieces. He purposely left the structures general, without any code notation, to allow other coders to implement the pieces in different programming languages. Three other leading code artists—Jared Tarbell of Levitated (http://levitated.net/), Robert Hodgin of Flight404 (http://flight404.com/), and William Ngan of Metaphorical.net (http://metaphorical.net/)—created the 26 code implementations. The project can be viewed at http://artport.whitney.org/commissions/softwarestructures/. When Reas is not lecturing around the world and developing Processing, he teaches as an assistant professor in the department of Design/Media Arts at UCLA. Reas is represented by bitforms gallery in New York, (www.bitforms.com), the BANK gallery in Los Angeles (www.bank-art.com/index.html), and the [DAM] in Berlin (http://dam.org/). His work can be viewed online at http://reas.com/.

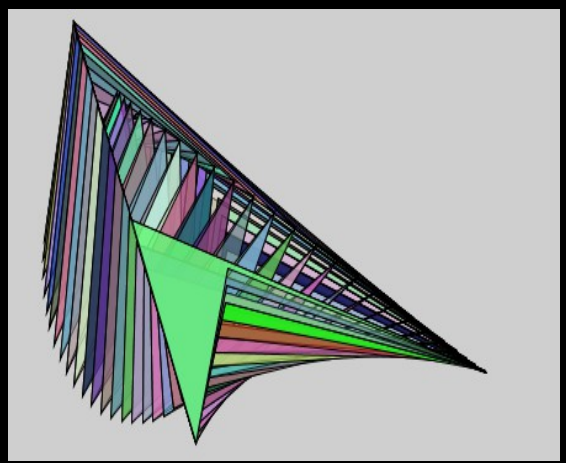
Chapter 3: Theory of Recreational Coding

The theory of recreational coding is based on these primary principles:

- Enable the skill developer to acquire amazing skills in art or code art using the amazing language.
- Experience digital exercising of processing facility as a main source of pleasure and beauty.
- Account of recreational beauty and fascination available from the digital outputs of processing language.
- Proof creatives as a digital proves of recreational coding of code art and not a formal proof of digital creatives.
- Appreciate and enjoy digital (in)dependence of the results of processing facility.







3.1 How does Processing works?

Processing does work in static and active modes. A static mode of application programming of processing facility is shown with a code snippet below with the output:

```
int    x;
int    y;
float   outsideRadius = 150;
float   insideRadius = 170;
x =   width/  2;
y =   height/  2;

 int  numPoints = int(map(  mouseX, 0, width, 6, 60));
 float     angle=0,         angle1 = 0;
 float   angleStep = 180.0/numPoints;

 beginShape(PShape.QUAD_STRIP);
 for (int i = 0; i <= numPoints; i++) {
  float px = x + cos(radians(angle)) * outsideRadius;
  float py = y + sin(radians(angle)) * outsideRadius;
  angle += (angleStep);
  fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
  vertex(px, py);
  px = x + sin(radians(angle1)) * insideRadius;
  py = y + sin(radians(angle1)) * insideRadius;
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
  angle += (angleStep);
 }
 endShape();
```

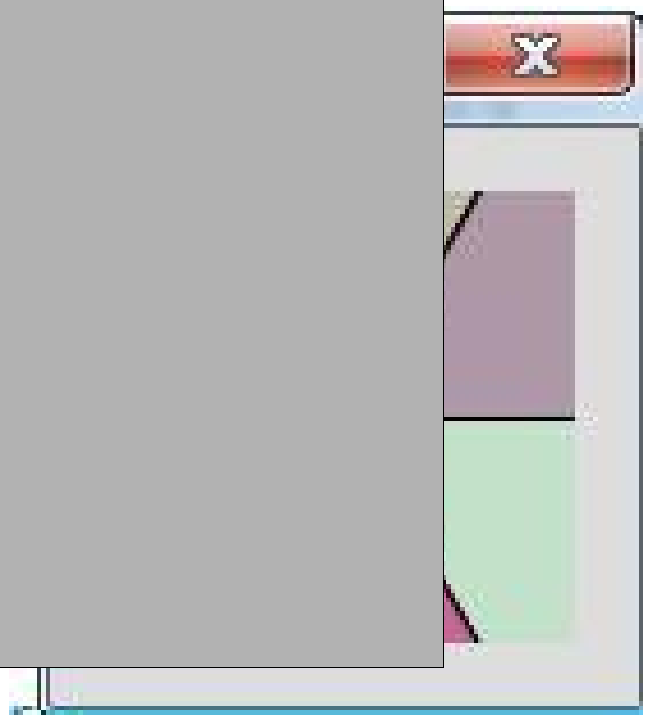


Figure 5: Rectangle Color Division

It is a simple application which draws a rectangle with six divisions. The application UI interface shows a run icon, a

minimization button and a close button. The rectangle is colored with six different colors in each division. There are two main methods of structuring a program in processing language namely: function-based or object-oriented-based. The function-based , also procedural calls processing facility methods from the facility library to be processed after been processed at the machine level language by the graphics cards. The second approach is the object-oriented structures which works just as the prior processing facility language, Java. Objects of the structures are compiled along the processing libraries to generate the creative graphics termed code art. Lets go over the program above.

```
Program : Rectangle  Color Division

Variables: x: int, y: int , numPoints:int , i: int;
          150= outsideRadius: float, 170= insideRadius: float, 0=angle: float, 0=angle1: float, angleStep: float
           px: float, py: float ;

Initializes: x= width/2, y= height/2;
initialize numPoints to the mouse point X value.  angleStep = 180.0 / numPoints.

{ BeginShape: QUAD_STRIP}
for  i=0  to numPoints do
begin
px= x + cos(radians(angle)) * outsideRadius;
py = y + sin(radians(angle)) * outsideRadius;

angle += (angleStep);

fill (color);

vertex(px, py) ;

  px = x + sin(radians(angle1)) * insideRadius;
  py = y + sin(radians(angle1)) * insideRadius;

 fill(color);

 vertex(px, py);

angle += (angleStep);

end

{EndShape}; {CREATE_GRAPHICS}
```

The program layout allows the reader to:

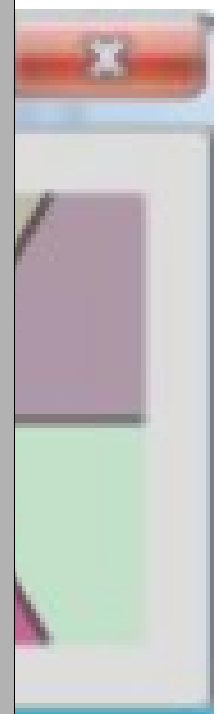
1) Specify a title of program.
2) Declare the variables used in the program.
3) Declares the initialization values for the variables.
4) Begin the shape creation portion of the program with computation. The graphics facility receives its instructions with the following procedures: fill(color), vertex(pointX, pointY).
5) Iteration of process 4 to the number numPoints times.

This program then only instructs the processing facility only two times and the rest of the program is the computation to create graphics. This is termed as *Aesthetics + Computation*. The aesthetics part is the fill and vertex program and the calculations of px, py and angle program is the computation part. Lets now look at the active mode of application programming in processing facility.

```
  int   x;   int   y;
  float   outsideRadius = 150 ;   float   insideRadius = 170 ;
  int  numPoints = int(map(  mouseX, 0, width, 6, 60));              //Line code added
  float     angle=0,         angle1 = 0;
  float   angleStep = 180.0/numPoints;
  x = width/2; y = height/2;

void setup() {                                                        //Line code added
  size(740, 550);
  background(255);
}                                                                     //Line code added

void draw() {                                                       //Line code added
  beginShape (PShape.QUAD_STRIP);
  for (int i = 0; i <= numPoints; i++) {
    float  px = x + cos(radians(angle)) * outsideRadius;
    float  py = y + sin(radians(angle)) * outsideRadius;
    angle += (angleStep);
    fill (random(0, 245), random(10, 245), random(20, 255), random(30, 245));
    vertex(px, py);
    px = x + sin(radians(angle1)) * insideRadius;
    py = y + sin(radians(angle1)) * insideRadius;
    fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
    vertex(px, py);
    angle += (angleStep);
  } endShape();
}                                                                   //Line code added
```



on

This code has the structured the program under two main methods namely: setup and draw. The processing facility has its own instantiation class which calls these methods in the main method, the entry of application running of Java applications. But with all these addition of lines of code, the processing facility produces or creates the same graphics or digital creative. The methods are void facilities in the processing library. We will create a picture of the program layout for the processing facility code art.

```
Program : Rectangle  Color Division Facility

Variables: x: int, y: int , numPoints:int , i: int;
          150= outsideRadius: float, 170= insideRadius: float, 0=angle: float, 0=angle1: float, angleStep: float
           px: float, py: float ;

Initializes:  x= width/2, y= height/2;
initialize numPoints to the mouse point X value.  angleStep = 180.0 / numPoints.

Procedure setup:
 size(740, 550);
 background(255);
Initializes: x= width/2, y= height/2;
End; {setup}

Procedure draw:
{BeginShape: QUAD_STRIP}
for  i=0  to numPoints do
begin
px= x + cos(radians(angle)) * outsideRadius;
py = y + sin(radians(angle)) * outsideRadius;

angle += (angleStep);

fill (color);

vertex(px, py) ;

px = x + sin(radians(angle1)) *  insideRadius;

py = y + sin(radians(angle1)) *  insideRadius;

 fill(color);

 vertex(px, py);

angle += (angleStep);

end

{EndShape}; {CREATE_GRAPHICS}

End; {draw}
```

The next program facility has some of the variables move into the void methods and lets look at the changes in the digital graphics.

```
  int   x;   int   y;
  float   outsideRadius = 150 ;   float   insideRadius = 170 ;

void setup() {                                                    //Line code added
  size(740, 550);
  background(255);
  x = width/2; y = height/2;
}                                                                 //Line code added

void draw() {                                                   //Line code added
  int  numPoints = int(map(  mouseX, 0, width, 6, 60));       //Line code added
  float     angle=0,        angle1 = 0;
  float   angleStep = 180.0/numPoints;

  beginShape (PShape.QUAD_STRIP);

 for (int i = 0; i <= numPoints; i++) {

   float  px = x + cos(radians(angle)) * outsideRadius;

   float  py = y + sin(radians(angle)) * outsideRadius;

   angle += (angleStep);

   fill (random(0, 245), random(10, 245), random(20, 255), random(30, 245));

   vertex(px, py);

   px = x + sin(radians(angle1)) * insideRadius;

   py = y + sin(radians(angle1)) * insideRadius;

   fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));

   vertex(px, py);

   angle += (angleStep);

 } endShape();

}                                                                 //Line code added
```

Figure 7: Spinning Colored Pie

The computation of x and y values are lines of code added to the *setup* method and are based on static variables *width* and *height* from the processing library. The setup method have *size* and *background* method calls from the processing facility library. The initialization of *numPoints* variable is based on the *map* method call from the processing library. The declaration and initialization of *angle*, *angle1* and *angleStep* are done in the *draw* method call. The draw method is processing facility library which is called several times with sleep time controlled by the *frameRate* method of the processing facility library. The frameRate method takes an integer argument of time dimension in units of milliseconds. The frame rate tells the processing facility how many times in millisecond delay that the draw method should be called or invoked during running. There are about four method callings from the processing library and two static variables usage. The *numPoints* variable is now the animated control variable because it determines the number iteration or loops to make in executing the fill and vertex functions and it changes value based on the position of the mouse.

```
Program : Rectangle  Color Division Facility

Variables: x: int, y: int , numPoints:int , i: int;
          150= outsideRadius: float, 170= insideRadius: float, 0=angle: float, 0=angle1: float, angleStep: float
           px: float, py: float ;

Initializes: all variables;

Procedure setup:
 size(740, 550);
 background(255);
x= width/2, y= height/2;
End; {setup}

Procedure draw:

{BeginShape: QUAD_STRIP}
  initialize numPoints to the mouse point X value.  angleStep = 180.0 / numPoints.
  float     angle=0,         angle1 = 0;
  float   angleStep = 180.0 / numPoints;

for  i=0  to numPoints do
begin
px= x + cos(radians(angle)) * outsideRadius;
py = y + sin(radians(angle)) * outsideRadius;

angle += (angleStep);

fill (color);

vertex(px, py) ;

  px = x + sin(radians(angle1)) *  insideRadius;
  py = y + sin(radians(angle1)) *  insideRadius;

 fill(color);

 vertex(px, py);

angle += (angleStep);

end

{EndShape}; {CREATE_GRAPHICS}

End; {draw}
```

```
 void draw() {
background(255);                    // Line of code added
 int numPoints = int(map(mouseX, 0, width, 6, 60));
 float angle=0, angle1 = 0;
 float angleStep = 180.0/numPoints;
beginShape(PShape.QUAD_STRIP);
 for (int i = 0; i <= numPoints; i++) {
   float px = x + cos(radians(angle)) * outsideRadius;
   float py = y + sin(radians(angle)) * outsideRadius;
   angle += (angleStep);
   fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
   vertex(px, py);
   px = x + cos(radians(angle)) * insideRadius;
   py = y + sin(radians(angle)) * insideRadius;
   fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
   vertex(px, py);
   angle += (angleStep);
 }
 endShape();
}
```

The position of the mouse is the essential thing to note here because it gives variations of numbers to used by the numPoints variable. The next program facility uses the same code snippet with some variables move here and there changing the output of the program. The draw method is the actionable part of the code art . The code snippet has a called *background* method in the draw facility. Without it, the window shows a ring as in Figure 8 with an octagon inside the ring digital graphics and the graphics from any mapped mouse input draws on the earlier, and it forms quite another nice digital creative. Just move the mouse in the window to see how the graphics changes to the mapped mouse input. It shows a beautiful animated color ring digital creative.

```
Program : Rectangle  Color Division Facility

// All sample except draw method has been changed

Procedure draw:

background(255);

{BeginShape: QUAD_STRIP}
  initialize numPoints to the mouse point X value.  angleStep = 180.0 / numPoints.
  float     angle=0,         angle1 = 0;
  float   angleStep = 180.0 / numPoints;

for  i=0  to numPoints do
begin
px= x + cos(radians(angle)) * outsideRadius;
py = y + sin(radians(angle)) * outsideRadius;

angle += (angleStep);

fill (color);

vertex(px, py) ;

  px = x + sin(radians(angle1)) *  insideRadius;
  py = y + sin(radians(angle1)) *  insideRadius;

 fill(color);

 vertex(px, py);

angle += (angleStep);

end

{EndShape}; {CREATE_GRAPHICS}

End; {draw}
```

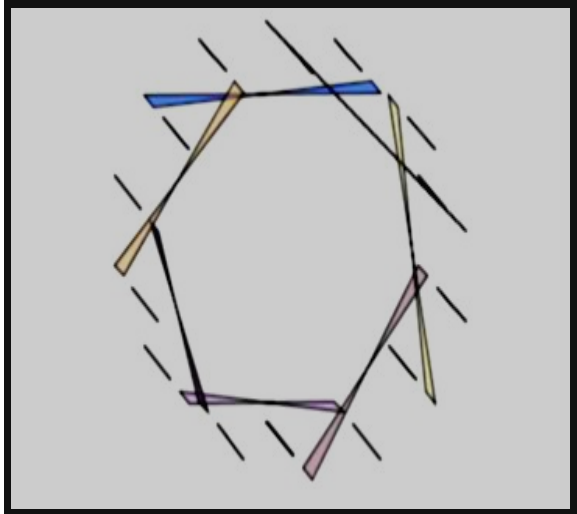
The program layout of the ring digital creative is shown in the side code picture. The *beginShape* method of the draw method for the digital quadring graphics is changed from QUAD_STRIP to TRIANGLE_STRIP and the graphics output is shown in Figure 9. The inside and outside radius is filled with several triangles instead of quad shapes.

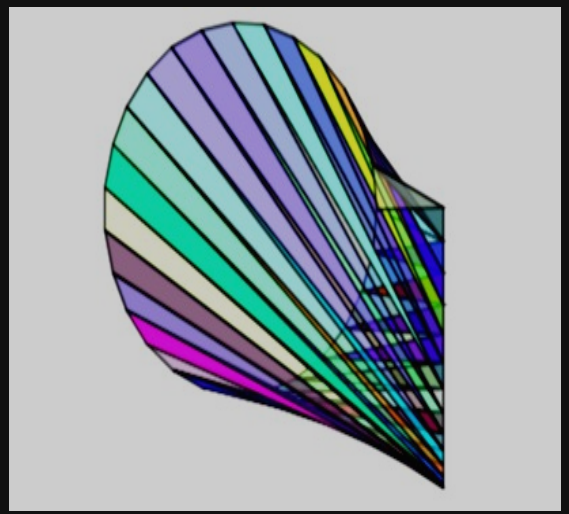
```
beginShape(PShape.TRIANGLE_STRIP);      // Line code has changes from
QUAD_STRIP to TRIANGLE_STRIP
 for (int i = 0; i <= numPoints; i++) {
   float px =x + cos(radians(angle)) * insideRadius;
   float py = y + sin(radians(angle)) * insideRadius;
   angle += (angleStep);
   fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
   vertex(px, py);
   px = x+cos(radians(angle)) * outsideRadius;
   py = y+sin(radians(angle)) * outsideRadius;
   angle += (angleStep);
   fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
   vertex(px, py);
 }
 endShape();
```

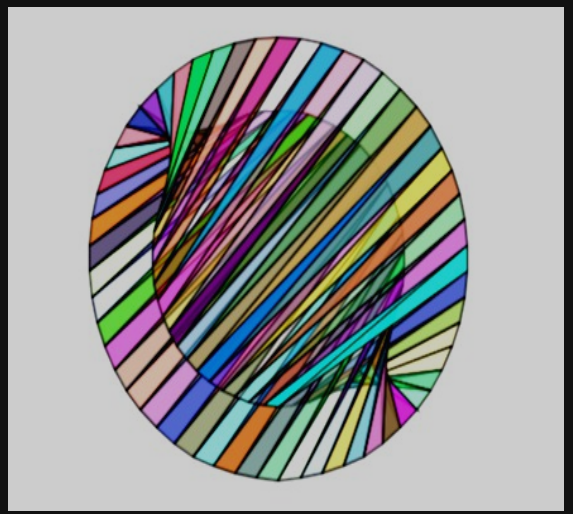
Figure 9: Creative Trirring [Strip Colored]

Chapter 4: Computer Graphics

Computer graphics is the synthesis of pictures of real or imaginary objects. The subject of computer graphics is introduced here with a list of examples that includes microprocessor-based video game processor, home computer, cartography, film or video recorder, simulation and animation, storage tube display terminal and plotting. A brief history of computer graphics is discussed.







Computer Graphics

4.1 Introduction

Computer graphics is the creation, storage and manipulation of models of objects and their pictures via computer. It is also the synthesis of pictures of real or imaginary objects. Graphics and visualization are one of the research interest of computer science. The subject of computer graphics is introduced here with a list of examples which will be familiar to most readers:

- *Microprocessor-based video game processors.* With the simple type, one attaches to a home TV a black box that can generate crude text and drawings and sample the levers or paddles used as interaction devices. A non-exhaustive list includes games such as Space Invaders, Asteroids, PAC-Man and Missile Command, test skills such as piloting and target-shooting via animation and changes to shape and color effects.
- *Hobby (home) computers*. With either a black and white or color TV set as a display, used for both text and plots, for playing games, etc. The pictures typically have low resolution: the individual dots which make up the characteristics, lines and solid areas can easily be seen.
- Storage tube display terminals with keyboards and a screen cursor controlled by thumb-wheels, used as alphanumeric time-sharing terminals as well as for interactive plotting of text and line drawings. Whenever a new picture is to be drawn, the screen is erased with a flash and a new picture is generated, often a character or line at a time.
- *Film or Video Recorder* for producing high-quality realistic-looking color pictures of real or imaginary objects for slides, movies, or TV. Examples are business graphs, logotypes for the TV network and scenes from science fiction epics.
- *Drum or flatbed plotters* used for generating line drawing such as 2D (two-dimensional) and 3D (three-dimensional) graphs, pie charts, histograms, flowcharts, architectural diagrams and circuit layouts.
- *Plotting* in business, science and technology. Graphics today is probably still most frequently used to draw 2D or 3D graphs of mathematical, physical and economic functions.
- *Cartography*. Computer graphics is used for the production of highly accurate representations on paper or film of geographical and other natural phenomena. Examples include geographic maps, relief map, contour map, oil exploration maps and population density map.
- *Simulation and Animation*. Computers produced animated movies of the time-varying behavior of real or stimulated

objects are becoming increasingly popular. Cartoons have a very high visual quality and the simpler kinds are becoming cost-effective.

4.2 History of Computer Graphics

While the future of digital creative as the standard means of user-computer creation is rosy indeed. It is important to understand where the subject topic is today and how it arrived at present state. Thus, this book is concerned with the techniques that were derived in the creation of digital creative. It is easier to chronicle the evolution of output technology and input technology or hardware and software that will have influence on the way the topic will develop. Thus, it is the hardware that we will begin with it.

4.2.1 Output Technology

The display devices developed in the mid-sixties and still in used today are called vector , stroke or calligraphic displays. They consist if a display processor, a display buffer memory, and a CRT with its associated electronics. The buffer stores the computer produced display list or display program; this contains point and line plotting commands (with coordinate endpoint data) and character plotting commands. These commands for plotting points, lines and characters are interpreted by the display processor; it converts digital values to analog voltages which displace an electron beam writing on the phosphor coating of the CRT. Since the light output of the phosphor decays in tens or at most hundreds of microseconds, the display processor must cycle through this list to refresh the phosphor at least 30 times per seconds to avoid flicker; hence the buffer holding the display list is usually called a refresh buffer. In raster displays, the display primitives such as lines, characters and solid areas (typically polygons) are stored in a refresh buffer in terms of their component points called pixels or pels (short for picture elements). The image is formed from the *raster*, a set of horizontal raster lines each made up of individual pixels; the raster is thus simply a matrix of pixels covering the entire screen area. The entire image is scanned out sequentially, 30 times per second, one raster line at a time top to bottom by varying only the intensity of the electron beam for each pixel on a line. The storage needed is thus greatly increased in that the entire image of say 128 lines of 128 pixels, must be stored explicitly in a

bit map containing only points that map one-for-one to points on the screen. The development that made raster graphics possible was that of inexpensive solid-state memory.

4.2.2 Input Technology

The development of output technology was done in parallel with the improvement of input technology. The fragile light pen has largely been replaced by a thin stylus moved on a data tablet or by a transparent, touch -sensitive panel mounted on the screen. Audio communication holds much exciting potential for the future of hands-free input and natural output of simple instructions, feedback, etc. With this devices much information can be drawn to the operation of picture components to be operated upon.

4.2.3 Graphics Formats

In computer graphics, graphic image types are generally broken down into two basic categories: raster graphics and vector graphics. **Raster graphics** are usually continuous tone images, like photographs, that are not delineated into precise shapes with smooth edges, but are built up as a series of pixels, like a painting. The pixels blend together to form the illusion of a single image. The BufferedImage class is a structure designed to handle raster graphics. **Vector graphics** are usually simpler images, in terms of number of colors and detail. They have distinct parts, each delineated with a concise edge. While raster graphics are a matrix of the actual pixels displayed on the screen (or an approximation), vector graphics are mathematically described shapes, with color fills, strokes, and other rendering attributes. Vector graphics can require much less memory compared to raster graphics, as they represent the plots of mathematical expressions. Ultimately though, in the rendering process to the screen, eventually everything must be converted to screen pixels. However, before the screen rendering, vector graphics are able to be stored as simple expressions, rather than large arrays of RGBA color values, as with raster graphics.

4.2.3.1 Raster graphics

Raster graphics are typically utilized for photographic or other highly detailed continuous tone images, where edges are not distinct and there is a large variation in color or value. Common raster graphic image file formats include TIF, BMP, and JPG. Raster formats also generally utilize some compression algorithms to ease their memory storage requirements. Compression can be **lossy**, in which the image quality is actually degraded with compression, like in JPG; or **lossless**, with no image loss, as in LZW compression, used commonly with the TIF file format. Raster images have variable resolution, meaning that you can specify how many dots of image information you want to pack into each inch. The more dots per inch (dpi), the finer the image detail or resolution.

4.2.3.2 Vector graphics

Vector graphics are typically used for simple illustrations, logos, and images that require distinct and precise contours. Computer text is vector-based, normally just wrapped in a text-based object that includes certain methods for editing the text. In vector-based graphics applications like Illustrator, FreeHand, and Flash, you can create outlines from text objects, or break them apart and get down to the vector information. Vector graphics are made up of vertices (points) and strokes. The pixels contained within the bounds of the vector shapes do not need to be stored in memory as they would for raster graphics. Vector graphics are also resolution independent.

4.2.4 Animations

In the simplest sense, animation is a series of still images displayed sequentially. If the rate of change between images, commonly called the **frame rate**, is fast enough, our brains can't detect the transitions. This effect is referred to as persistence of vision. In actuality, we perceive the physical world in much the same way. Light enters our cornea, moves through the pupil and lens, and eventually reaches the retina. The retina is the light-sensing part of the eye, containing photosensitive structures called rods and cones. The rods detect low light, while the cones detect color and detail. The received light is transformed, through a chemical process, into electrical impulses, which then pass through the optic nerve into the brain to form vision. Thus, our vision is not static, but—similar to animation—a perpetual sequence of dynamic 2D data. Animation is really just an

extension of static imaging.

4.3 Digital Creativity

The conceptual framework on which to base digital creativity is called *CreativePlanet*. The hardware component is the host computer that drives the display device, *monitor* and an input component or interactive component, *keyboard* for entering text. The *software* consists of three components. The application program; it stores in memory and retrieves from the second component, application data in the *data file system* and sends graphics to the third component, the graphics system.

Framework :CreativePlanet

The framework of *CreativePlanet* is based on the principles of

1. Application Window : The application window is a screen-based window with several screens that can be changed from the keyboard by entering a key from the menu display. The application window enables the application to be close down or minimized.
2. Drawing : It's purpose is to draw (draw method of processing) all the creative graphics in the *CreativePlanet* framework. A typical wish of a picture of a digital creative displaying on the window of application is handled by just a draw methodology. A draw method describes the image object to the graphics device or card and then it display it to the screen of the application window driven by the display device. The drawing of creative graphics is just an extension of static imaging- A perpetual sequencing dynamic imaging. Thus, there is snapshot for each digital creative graphics set up.
3. Menu Displaying : A window displaying a creative graphics will need a capability to switch or change between screen without stopping or closing application running. With this in mind, then the menu display is a window for displaying a number of interactive instructions for the creative user to have the ability to move from screen to screen.
4. Text Graphics : It is the means to display instructions for each screen of the application window. It describes the activity taking place for the changing screens. There is an animated text facility for the *CreativePlanet* framework. There is an outline from the fixed-typed text objects in the application program.
5. Planet World : It is a physical world stimulated by a number colored ball depicting a planet each and floating on the

screen of application window. There are two types of world namely planetx world, where a planet (depicted as a ball) moves from the left of application window to right, then it re-appears after it enters the right-end window and planet universe is made up of several planets with a fixed-animated section-screen and a floating billboard.

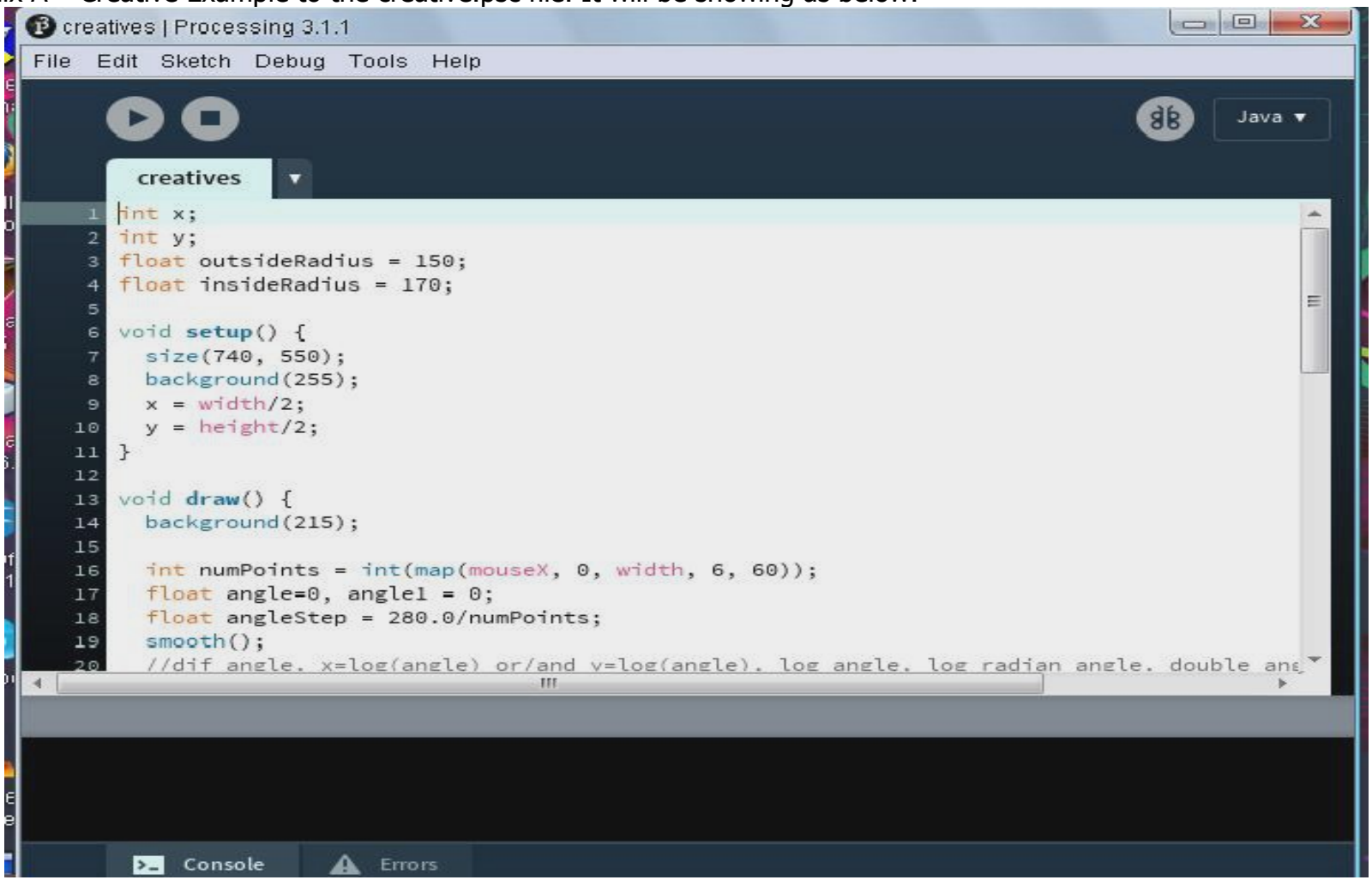
6. Interactive Command : The display instructions on the menu screen is a set of interactive commands for the application framework. The commands are interactive because of the interactive input device, keyboard and of course responsive to cause the changing screens. With the interactive commands, a creative user can press for a different screen of application window each time. CreativePlanet framework is an interactive paradigm that is natural for a graphics representation.

```
Program: CreativePlanet
PImage img;;                    // Image Object
int    i =1  , l =1;                    // New Image Counter s
float    ci=1  ;
String   path,   path1,    path2;
int     textSize  =40;
color    c;
void   setup()   { ….....................................................}  // …..... Program Details Hidden
void   draw()    { ….....................................      // …..... Program Details Hidden
flashComparator();                                  //  Flash Manipulation
 project() ;                                        // Project Manipulation
 blinkComparator();                                 //Blink Manipulation
evenOrOddComparator();                              // Even or Odd  Comparator
 dualEffect();                                      // Dual Effect
 sectionComparator();                               // Section Comparator
 peom();                                            // Text Display
 menu();                                           // Menu Display
 image( img );                                                  // Image Display
 text("FASCINATION ");                              // Displaying   FASCINATION
}
{CreativePlanet}
```

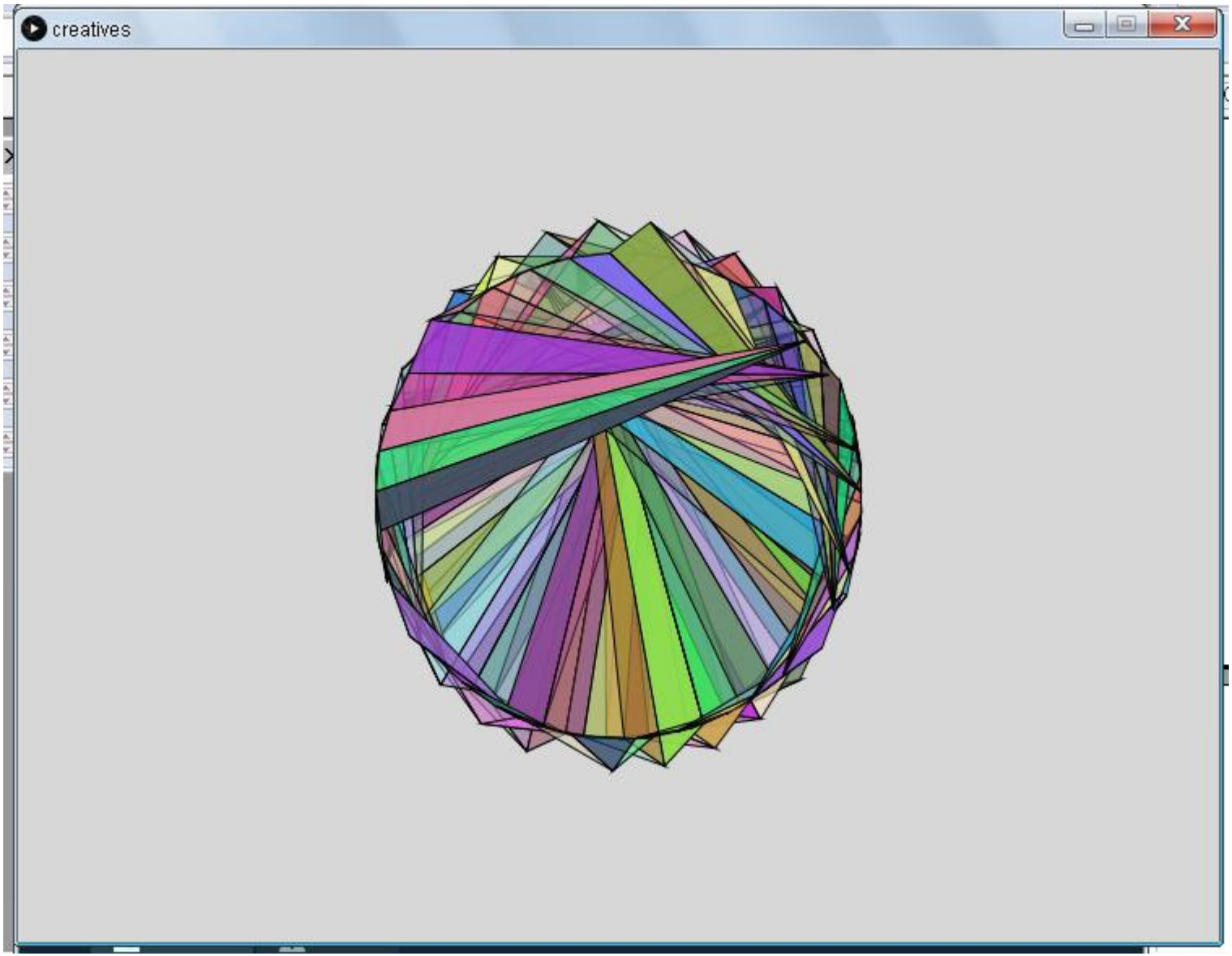
The CreativePlanet framework has a program layout as shown above. The complete source code for the framework is shown in Appendix A – Source Code.

4.3.1 Creation of Creative Graphics

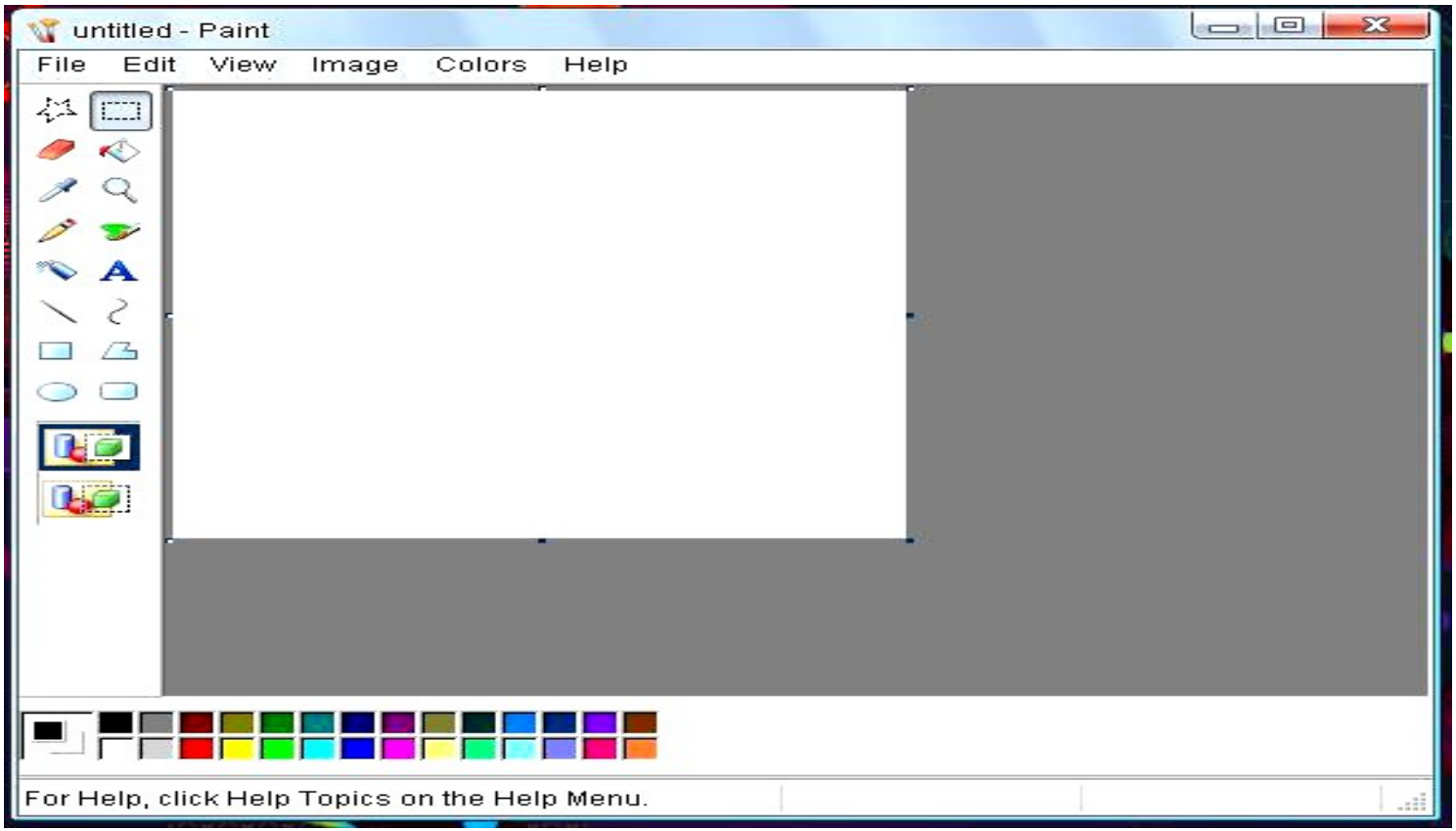
In this section, we will be creating the digital creatives and its pictures for the CreativePlanet program. We will start by first going through the creativeExamples folder then clicking on the creative.psc or create a text file and name it as creative.psc. You can get this from the books page at http://createspace.com estore and search for Digital Creatives or just copy from the Appendix A – Creative Example to the creative.psc file. It will be showing as below:



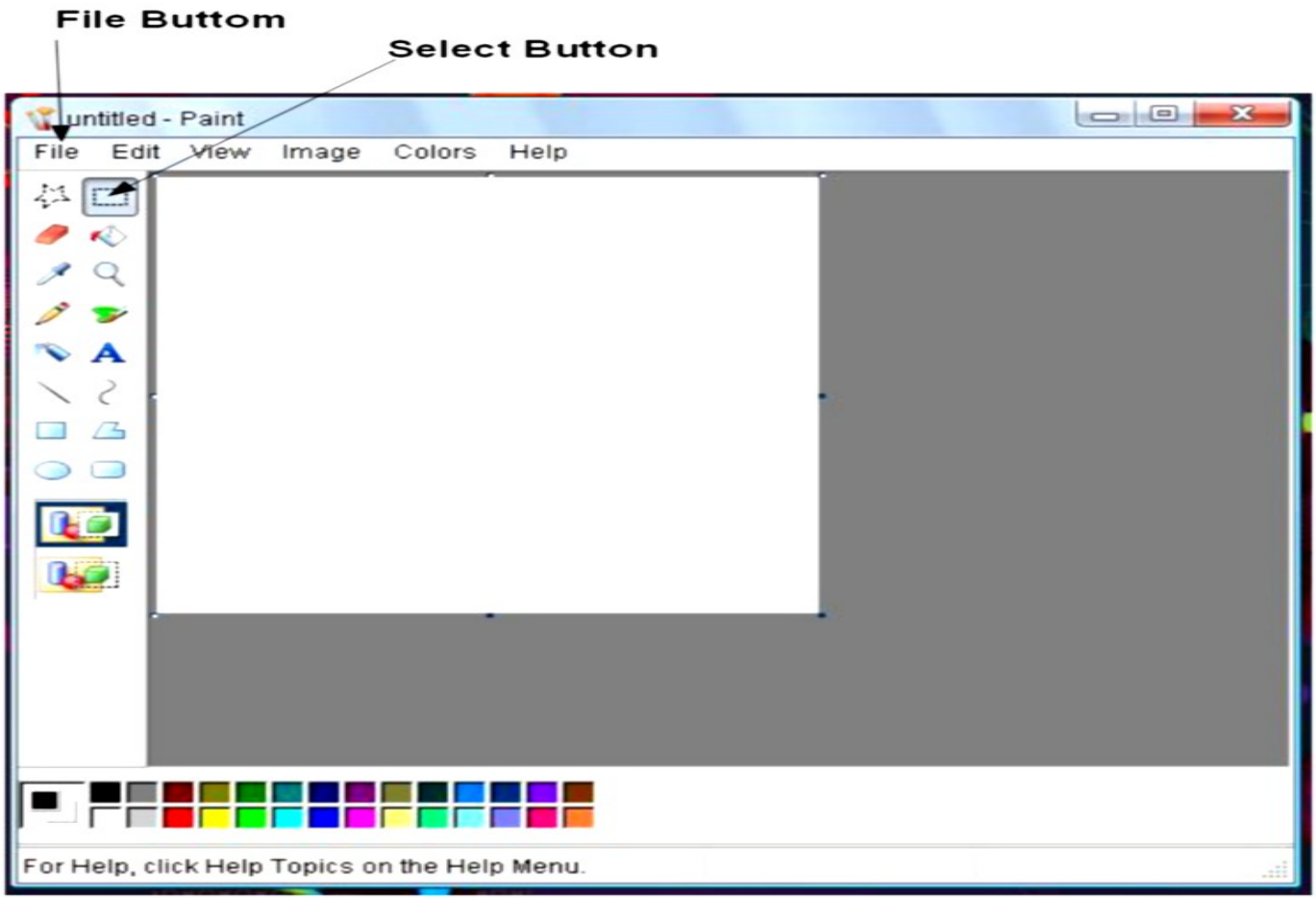
Now, run the application by clicking on the Run button in the processing IDE (Integrated Development Environment) and the



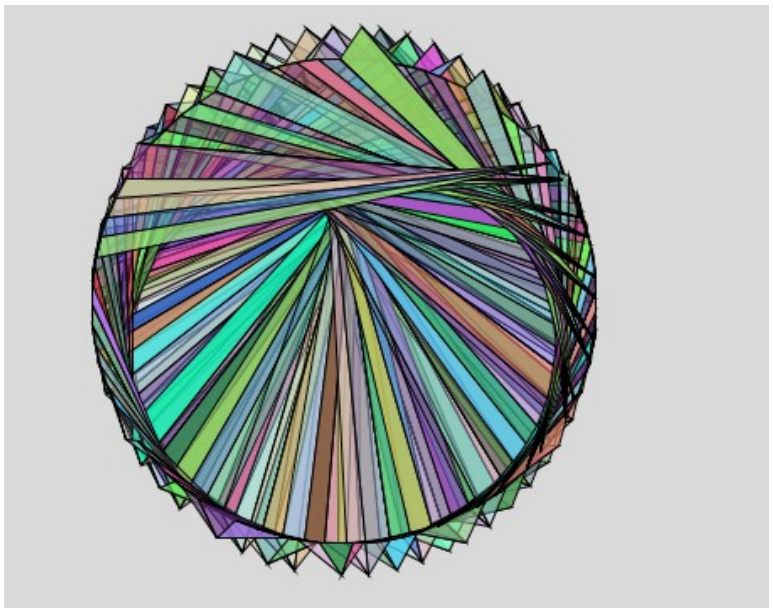
application window show as above. Move the mouse a bit if you do not want the creative graphics showing and then press you *Print Screen – SysRq* to get the digital creative into the clipboard. Then paste the clipboard content onto a new paint tool application screen. The paint tool on windows should be showing as below:



Then click on the "select button" to be able to crop the image to the selected dotted rectangle interest as shown below:


File Buttom
Select Button
untitled - Paint
File
Edit
View
Image
Colors
Help
For Help, click Help Topics on the Help Menu.

After the digital creative graphics is painted in the paint tool, then click on the File ► Save Button to save the creative graphics to a proper file on you host computer. It is much preferred to save to jpeg than any other graphics file type; png, bmp, gif, tiff etc. Compression can be **lossy**, in which the image quality is actually degraded with compression but it is fine. The final digital creative graphics will look like:



4.3.2 Storage of Creative Graphics

The creation of the creative graphics is done and it is important to keep the digital creatives at better store on your host computer by either compressing with WinRar to either .rar or .zip (default in windows). You can also save the digital creatives at Google Drive or Cloud Storage and those are for more professional users. It can be printed onto materials or papers also as a means of storage and a personal electronic publishing server like xammp, gallery, etc. If you are on the fantasy side then you use the social media like Facebook, Google+, Instagram, etc.

4.3.3 Manipulation of Creative Graphics

The manipulation of the creative graphics is about the different programming methodology that can be used in the computer graphics perspective. It based on the CreativePlanet framework and program source code in Appendix. In here, the program layout will be depicted and explained further. There are three main categories of creative graphics used in the CreativePlanet program: original group, second creative group, small group. The small group is just the same as the second creative group but will display with a smaller size. The manipulation of the creative graphics are:

1. Flash Comparator
2. Projection
3. Blink Comparator
4. Even Comparator
5. Odd Comparator
6. Dual Effect
7. Section Comparator
8. Text Animation
9. Look Component

We will start with the flash comparator and the rest will follow.

1. Flash Comparator

The flash comparator method, *flashComparator* is a method of the CreativePlanet program .The method layout is shown below:

```
Method: Flash Comparator
Return : void
if  (original)  {
clear();
 background(color(33, 3, 3, 3));
   path="data/creative3 ("+i+").JPG";
   img=loadImage(path, "jpeg");
   x=140;
   y=180;
}
else if(second_creatives)
{  ….....................................
 path1="data/creative2 ("+l+").PNG";
.....................................
} else if(small_group) {
….....................................
 path2="data/creative2 ("+l+").PNG";
}
{Flash Comparator}
```

First, it checks if the original flag is set to true and then functions by clearing the paint panel, then changes the background, the path for the original group is assigned to *path* variable and the position of the image to be displayed is changed to ***position***(*width/2-x, (height/2-img.height)/2+y) .* If the second creative group with invert flag is set to true then it does the same as the original flag-if (pronounced: flagife) but only changes the path to the second category group. The third small_group flag when set to true will

function as the same as the first and second flags and with the path changed to it's right path on file directory “data/creative2 (I).PNG”, where I is the picture counter. The original path is “data/creative3 (I).JPG”. With this counter, the program continuously flash the digital creatives by looping through the digital creatives and changing as the counter increases.

2. Projection

The projection method, *project* method is a method of the CreativePlanet program. It functions by first blinking the screen and blending the image of the digital creative with a dummy image made up of a generated color. The types of blending manipulation or effect are dark, light, hard light, multiply and overlay.

Method: Project

Return : void

```
blink();
c=color(random(i, 255), random(i*random(2, 125)), random(i+random(10, 255)),
random(i*random(3, 20)));
PImage img1=createImage(i*2, i*3, i+10);
img.blend(img1, 0, 0, img1.width, img1.height, 6, 0, 33, 100, f);
background(c);
```

{**Project**}

3. Blink Comparator

The blink comparator method, *linkComparator* method is a method of the CreativePlanet program. It changes the text size for the text display, it blinks the screen by changing the background color, thirdly it performs a look calling with the *look* method and finally does a comparison, *comparator* method with variable *i* and *l* increments. The *comparator* method works by clearing the background for first, then sets the original flag to true, then performs a look again, assigns the image from the *look* function and then flags the original variable to false. After all of these, it sizes the image from the look function and finally paints the image on the panel component.

Method: Blink Comparator

Return : void

```
textSize  =20;
blink();
look();
comparator();
i++;
l++;
```

{**Blink Comparator**}

4. Even or Odd Comparator

Method: Even Or Odd Comparator

Return : void

```
clear();
blink();
origpic=true;
look();
PImage evenImg1=img;
origpic=false;
invert=true;
look();
```

{**EvenOrOddComparator**}

The even or odd comparator method, *evenOrOddComparator* method is a method of the CreativePlanet program. It clears the background of the paint component, then blink screen, sets the original flag to true, it then performs a look and assigns the image from the look function. After, it sets the original flag to false and sets the invert flag for the second category group to true, then performs a *look* function again, assigns the image from look and finally blends the two images then draws the image by painting it. The even and odd comparator of the manipulation is one function and does has reduced the number of items.

5. Dual Effect

The dual effect method, dualEffect method is a method of the CreativePlanet program. It clears the background of the paint component, blinks the screen of the application window, performs a look function, then compares with the *comparator* method, clears again, sets the original flag to true, then performs another look function, assigns the image from the second look, then turns the invert flag to true, another look, it sizes the image from the last look function, changes the background color and finally draws the last image by painting on the component along with dual image randomly placing along the top, bottom, right and left respectively. The dual image is the last image but a re-size image and positioning the image on the cardinal(top, bottom, right and left) sides of the last image.

Method: Dual Effect

Return : void

```
blink();
look();
comparator();
clear();
origpic=true;
look();
PImage evenImg1=img;
origpic=false;
invert=true;
look();
img.resize(190, 140);
background(color(33, 3, 3, 3));
switch(rand)
```

{**DualEffect**}

6. Section Comparator

The section comparator method, *sectionComparator* method is a method of the CreativePlanet program. It blinks the background, calls the *comparator* method, gets the image from the comparator function then gets a pixel of size 30 by 30 and fills the draw or paint color. It rotates the screen of the application window by PI / picture counter. It draws an ellipse which represents the planet and then translates to the ellipse position. The rotation and ellipse draws repeats for about nine times to create that planet effects.

Method: Section Comparator

Return : void

```
blink();
comparator();
color fillPisels=img.get(30, 30);
fill(fillPisels);
rotate(-PI/i);
ellipse(planetx, planety, 120, 120);          //Ball floating
translate(planetx, planety+h1);
.....                                         // More of repeated code
```

{**SectionComparator**}

7. Text Animation

There is no method calls for this particular CreativePlanet program. It just reads line by line a text file from the *data* directory and displays each read text on the screen. It uses counters to display part of the text. So a word "FASCINATION" with a 11 counter will display the full text but with a counter of 6 will display the text "FASCIN", then with counter 5 will display the text "FASCI". By this means the text animates and the changing read text data incrementing to animate and resets to zero to start over again. Sometimes, the text rotates on the screen of the application window. The text animations program layout is as below:

Draw: Text Animation

Return : void

```
if (fc<19) {                                          //fc is the text counter
    text(roman, 0, fc, width/4, 50);
    fc++;
  } else fc=0;
  rotate (-PI/(2*i));
text(roman, 5, 11, random(width/4), random(50));      // roman text
```

{**Text**}

8. Look Component

Method: Look

Return : void

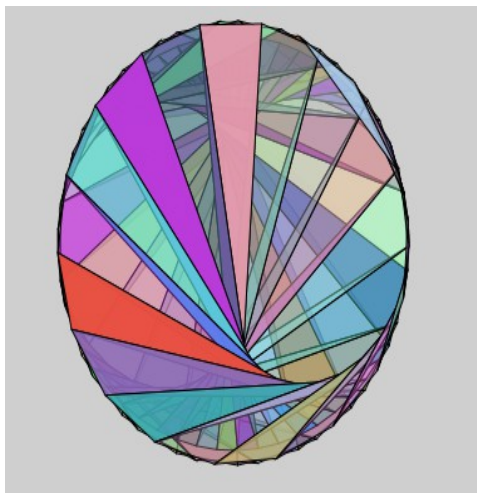
```
if (i>37 || i<=0) {
  i=1;
 }
if ( l <= 0 || l > 61)  {
  l=1; }
if (origpic)      {
  clear();
  background(color(33, 33, 33, 129));
  path="data/creative3 ("+i+").JPG";
  img=loadImage(path, "jpeg");
  img.resize(350, 250);
} else if (invert)   {  // perform similar instructions with path change  }
  else if (small)    {  // perform similar instructions with path change  }
```

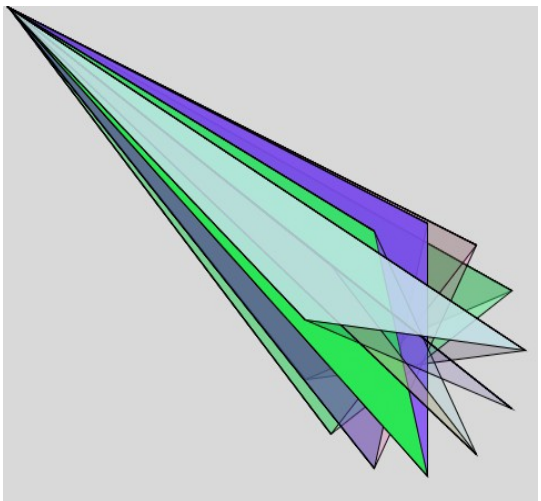
{**Look**}

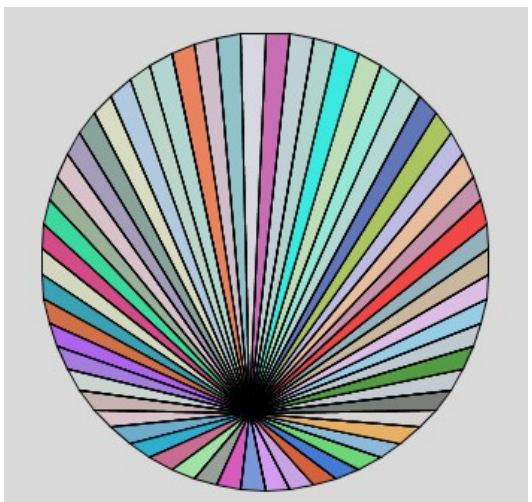
The look method, look method is a method of the CreativePlanet program. It checks the three flags of the three image categories and loads the image of the category if the flag is set to true in the execution. But first, it changes the background by setting it to a new color specified in the code. It does similar instructions for the other flags. The side program layout is detailed.

Chapter 5 Digital Creatives and Graphics Programming

This chapter looks at the digital creative in much more copy-run-display approach and the graphics programming to achieve the contents of this chapter. The contents of the chapter are all digital creative graphics and the code art to come to it. It chooses a workbook approach to the explaining of this text.







5.1 Introduction

It takes code art after the begin shape to end shape vantage points in the code art to describe the digital creatives and the graphics output is displayed in this text along with the code art. It is a copy-run-display workbook digest. In order to digital proof the graphics outputs just copy and run to display the digital creative for each *shape* code art. Each shape code art is assigned a number (34) for clarity and simplicity. Enjoy and have fun. It mapped input mouse is hard coded into the code art and that will be need also. Just follow the previous examples from Chapters 3 or 4 and the program layout can also help you to also figure it out. The program layout is constant and needs a closer attention to see that it is the same or similar code art. That is the great idea !

Shape 1

```
beginShape(PShape.TRIANGLE_STRIP);
 for (int i = 0; i <= numPoints; i++) {
   float px = x + cos(radians(angle1-angle)) * outsideRadius;
   float py = y + sin(radians(angle1-angle)) * outsideRadius;
   angle += (angleStep);
   fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
   vertex(px, py);
   px = x + sin(radians(angle1-angle1)) * insideRadius;
   py = y + sin(radians(angle1-angle1)) * insideRadius;
   fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
   vertex(px, py);
   angle += (angleStep);
   px = x + cos(radians(angle1)) * insideRadius;
   py = y + sin(radians(angle1)) * insideRadius;
   fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
   vertex(px, py);
   angle += (angleStep);
   px = x + cos(radians(angle1)) * insideRadius;
   py = y + cos(radians(angle1)) * insideRadius;
   fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
   vertex(px, py);
   angle += (angleStep);
 }endShape();
```

```
int numPoints = int(map(410, 0, width, 6, 60));
```

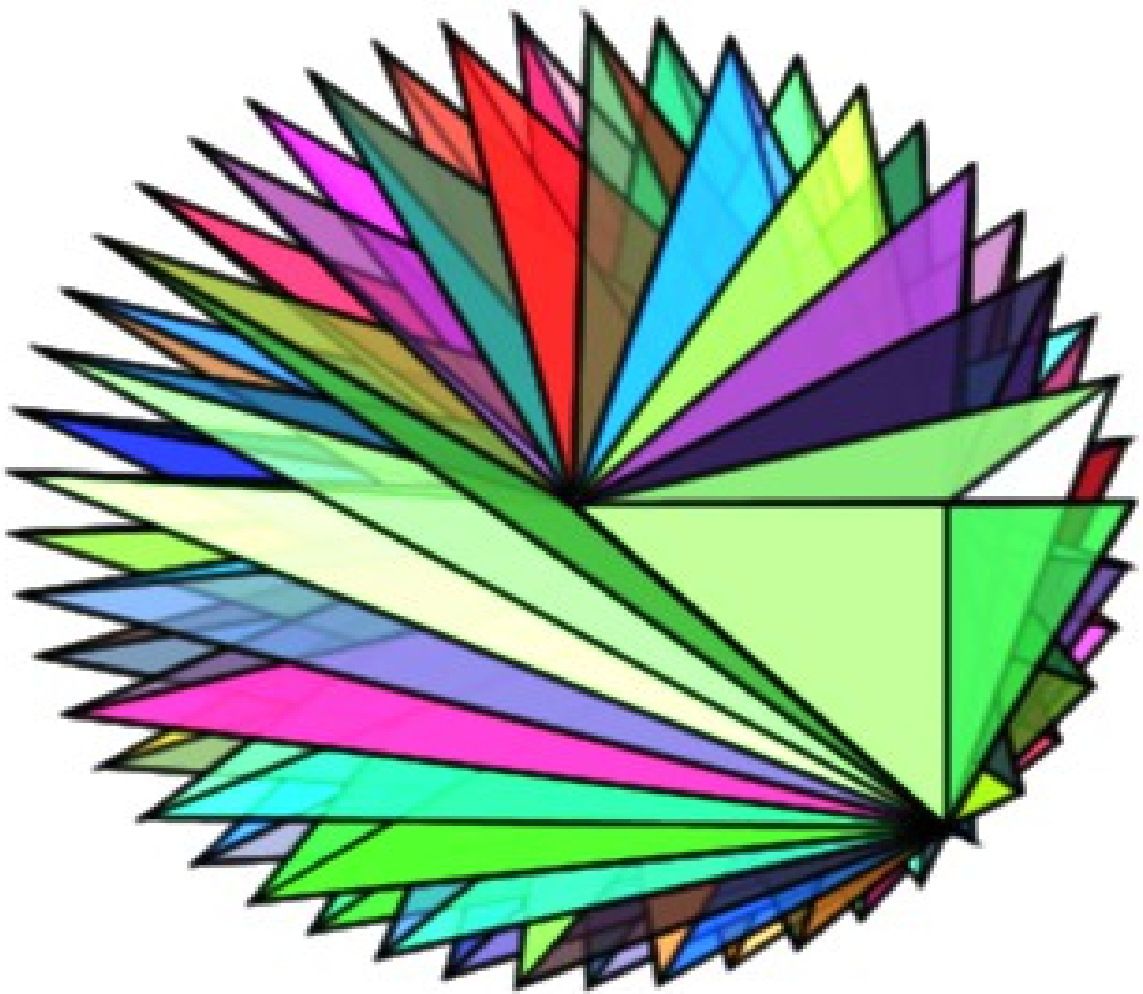


Illustration 1

```
int numPoints = int(map(12, 0, width, 6, 60));
```

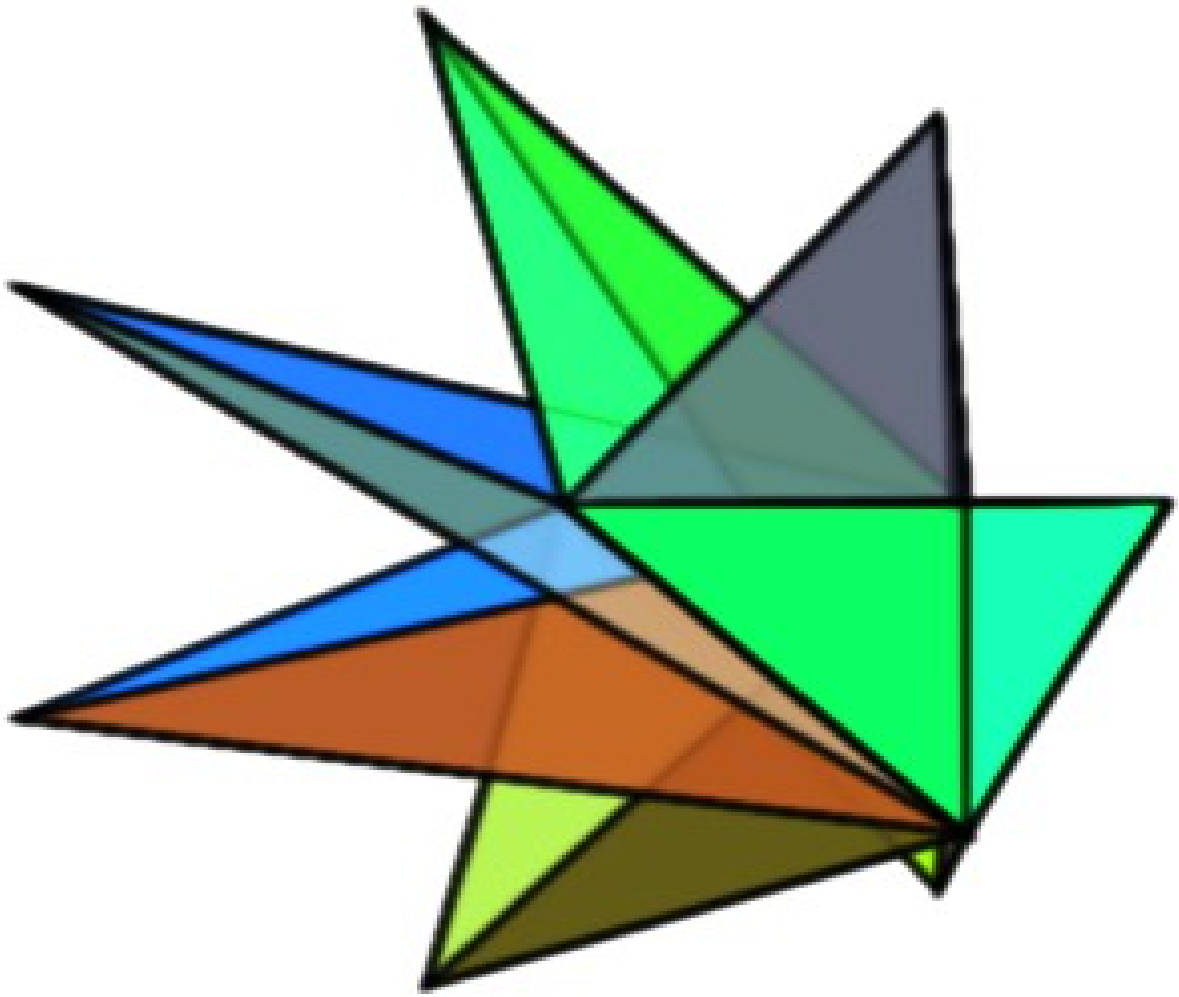


Illustration 2

Shape 2

```
beginShape(PShape.QUAD_STRIP);
for (int i = 0; i <= numPoints; i++) {
  float px = x + cos(radians(angle1-angle)) * outsideRadius;
  float py = y + sin(radians(angle1-angle)) * outsideRadius;
  angle += (angleStep);
  fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
  vertex(px, py);
  px = x + sin(radians(angle1-angle)) * insideRadius;
  py = y + sin(radians(angle1-angle)) * outsideRadius;
  //fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
  angle += (angleStep);
  px = x + cos(radians(-angle1)) * outsideRadius;
  py = y + sin(radians(-angle1)) * insideRadius;
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  //vertex(px, py);
  angle1 += (angleStep);
  px = x + cos(radians(angle)) * insideRadius;
  py = y + cos(radians(angle)) * outsideRadius;
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  //vertex(px, py);
  angle += (angleStep);
}
endShape();
```

```
int numPoints = int(map(426, 0, width, 6, 60));
```

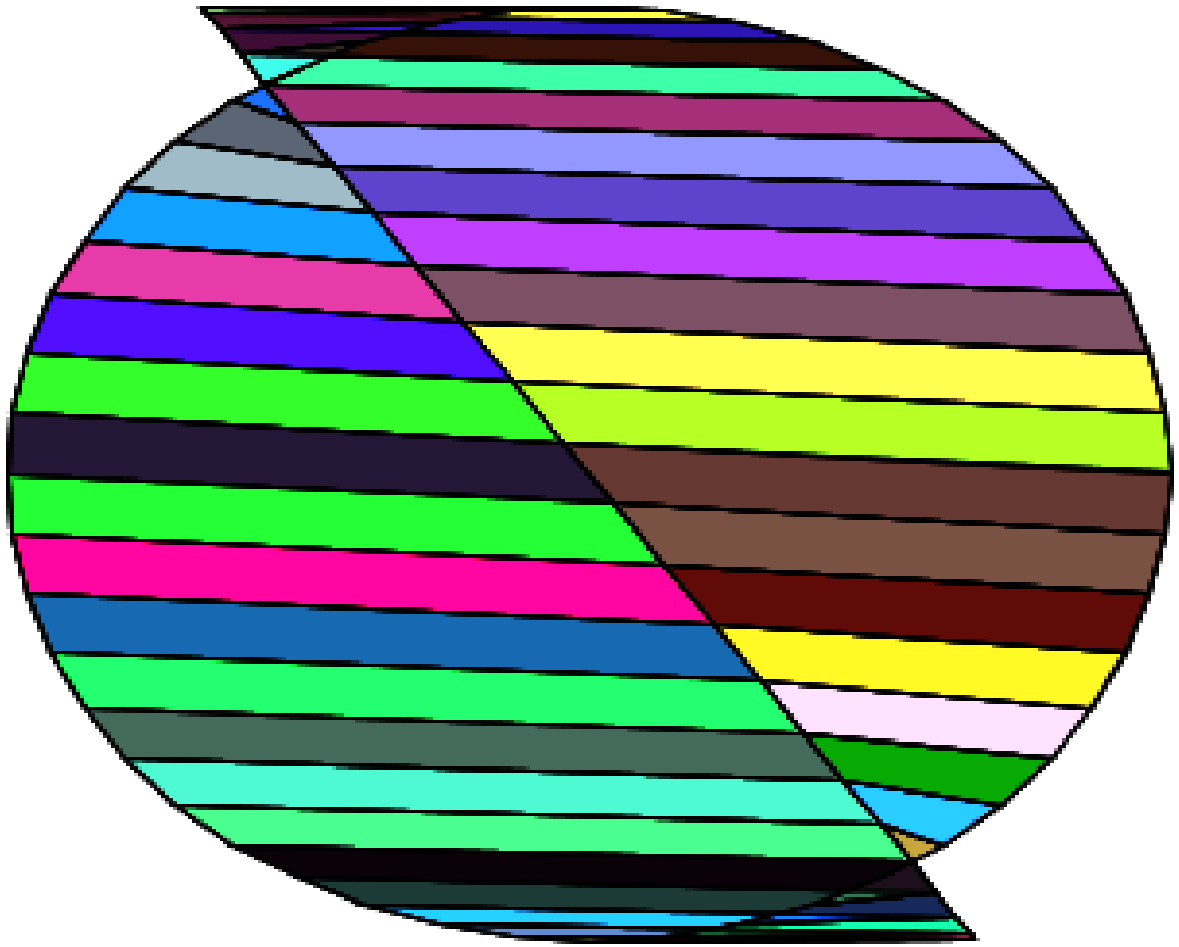


Illustration 3

Shape 3

```
beginShape(PShape.QUAD_STRIP);
 for (int i = 0; i <= numPoints; i++) {
  float px = x + cos(radians(angle1-angle)) * outsideRadius;
  float py = y + cos(radians(angle1-angle)) * insideRadius;
  angle += (angleStep);
  fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
  vertex(px, 0.9*py);
  px = x + sin(radians(angle1-angle)) * insideRadius;
  py = y + sin(radians(angle1-angle)) * outsideRadius;
  //fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(0.5*px, 0.9*py);
  angle += (2*angleStep);
  px = x + cos(radians(-angle1)) * outsideRadius;
  py = y + sin(radians(-angle1)) * insideRadius;
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(0.5*px, py);
  angle1 += (2*angleStep);
  px = x + cos(radians(angle-angle1)) * insideRadius;
  py = y + cos(radians(angle-angle1)) * outsideRadius;
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(0.5*px, 0.9*py);
  angle += (angleStep);
 }
 endShape();
```

```
int numPoints = int(map(500, 0, width, 6, 60));
```

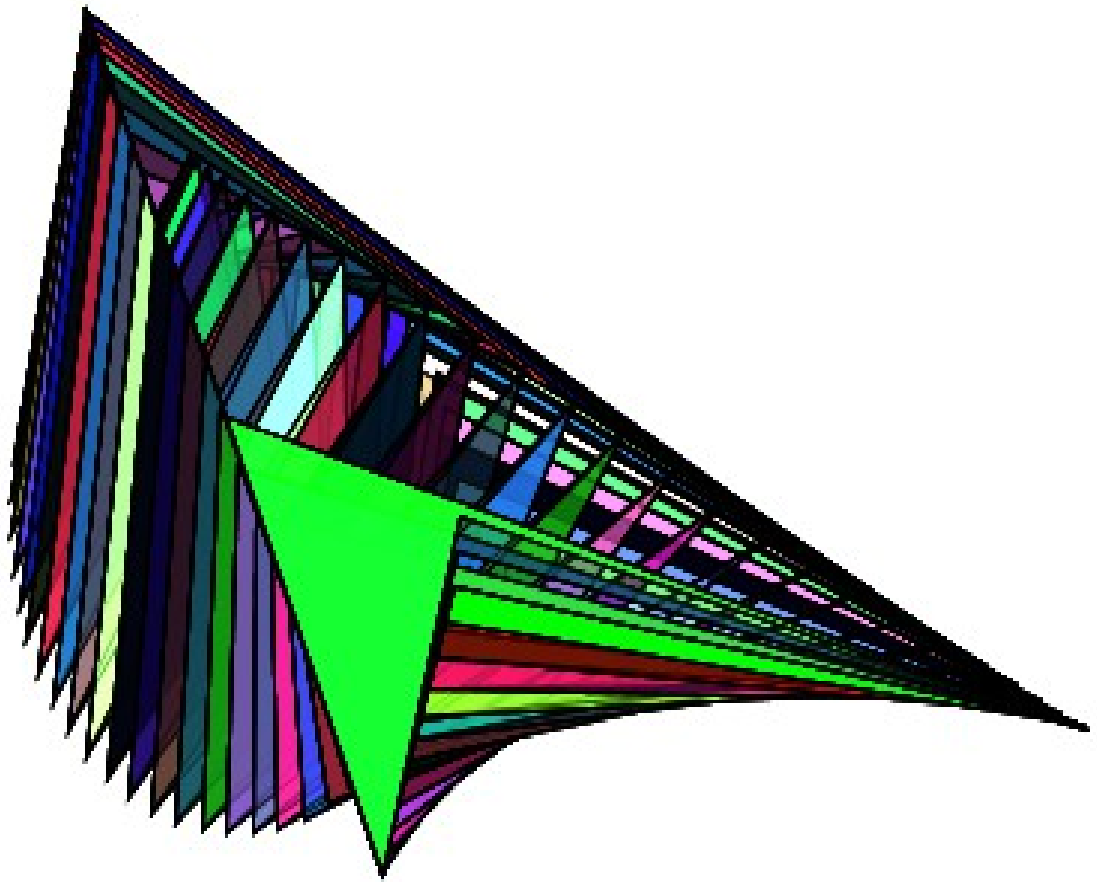


Illustration 4

Shape 4

```
beginShape(PShape.QUAD_STRIP);
for (int i = 0; i <= numPoints; i++) {
  float px = x + sin(radians(-angle)) * insideRadius;
  float py = y + cos(radians(-angle)) * outsideRadius;
  angle += (angleStep);
  fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
  vertex(px, px/py);
  px = x + cos(radians(angle1)) * outsideRadius;
  py = y + cos(radians(angle1)) * outsideRadius;
  //fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, px/py);
  angle += (2*angleStep);
  px = x + cos(radians(angle1)) * insideRadius;
  py = y + sin(radians(angle1)) * insideRadius;
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, px*py);
  angle1 += (2*angleStep);
  px = x + cos(radians(-angle)) * outsideRadius;
  py = y + sin(radians(-angle)) * insideRadius;
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, px*py);
  angle += (4*angleStep);
}
endShape();
```

int numPoints = int(map(300, 0, width, 6, 60));

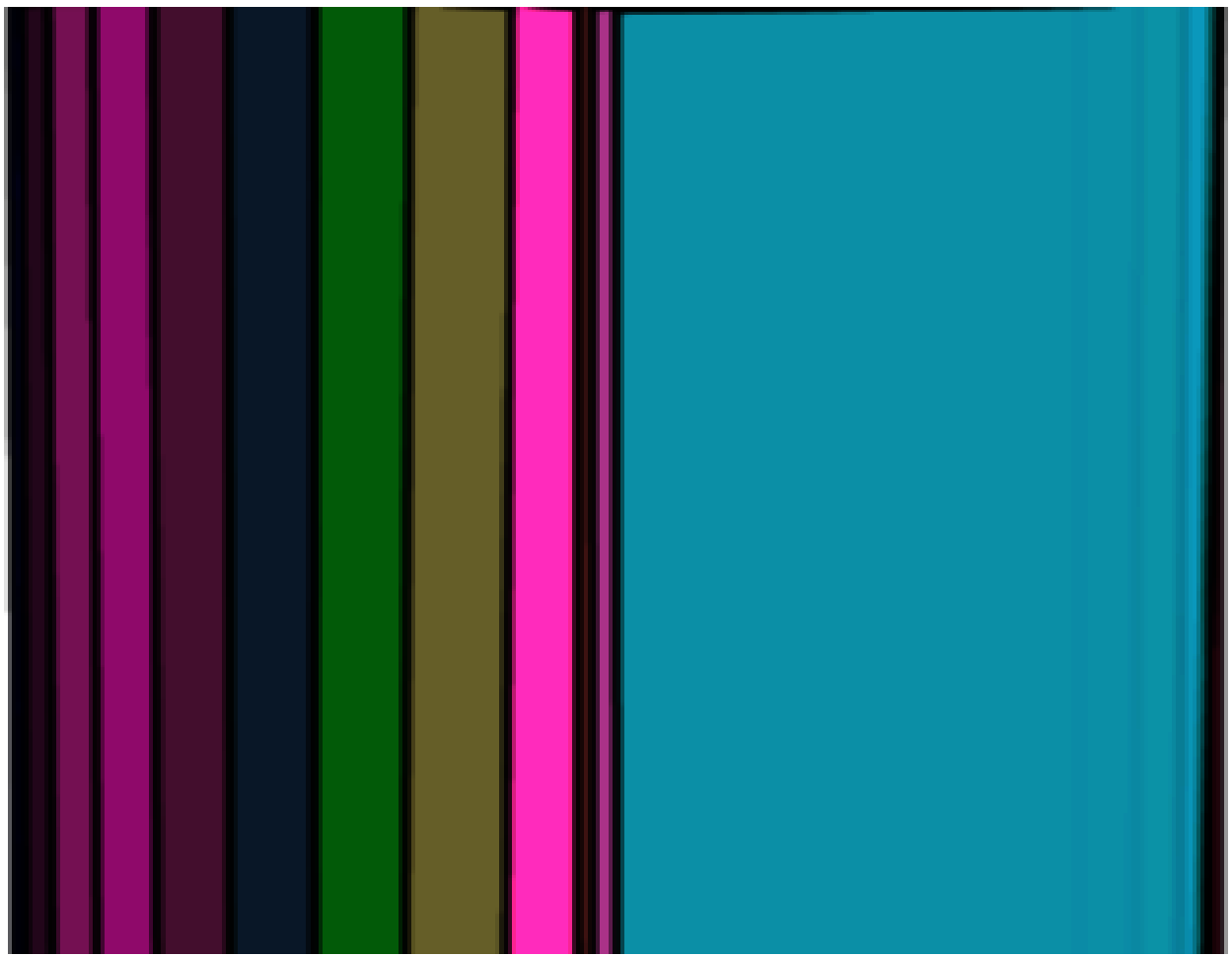


Illustration 5

Shape 5

```
float outsideRadius = 150;
float insideRadius = 10;
 float angleStep = 180.0/numPoints;

beginShape(PShape.QUAD_STRIP);
  for (int i = 0; i <= numPoints; i++) {
    float px = x + sin(radians(angle)) * insideRadius;
    float py = y + cos(radians(angle)) * insideRadius;
    angle += (angleStep);
    fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
    vertex(px, py);
    px = x + sin(radians(angle)) * outsideRadius;
    py = y + cos(radians(angle)) * outsideRadius;
    fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
    vertex(px, py);
    angle += (2*angleStep);
  }
  endShape();
```

```
int numPoints = int(map(34, 0, width, 6, 60));
```

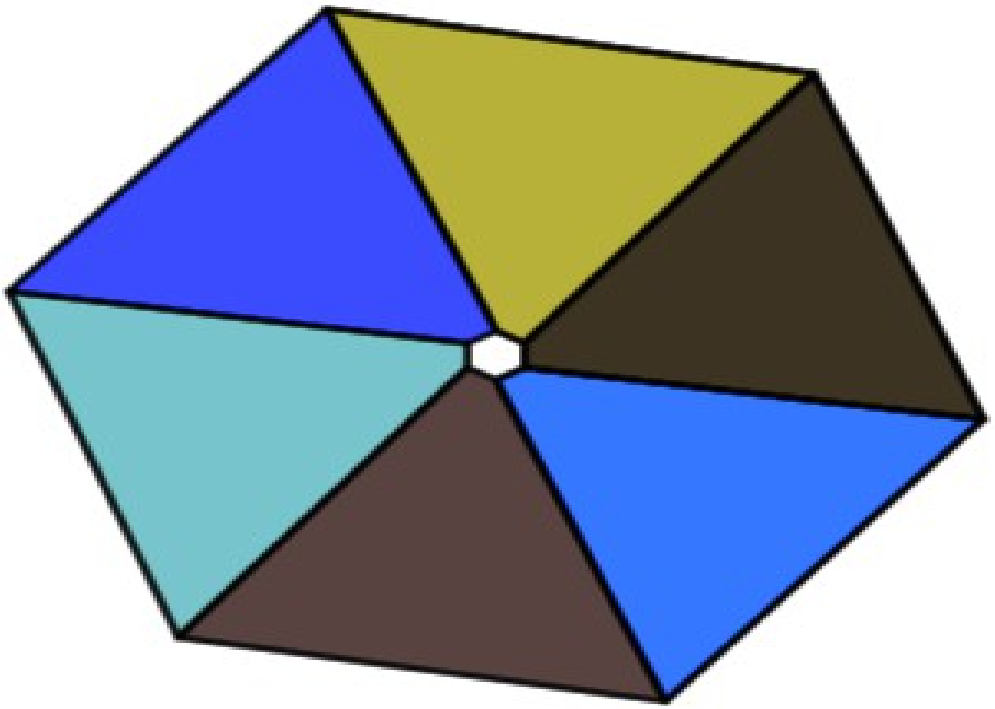


Illustration 6

```
int numPoints = int(map(8, 0, width, 6, 60));
```

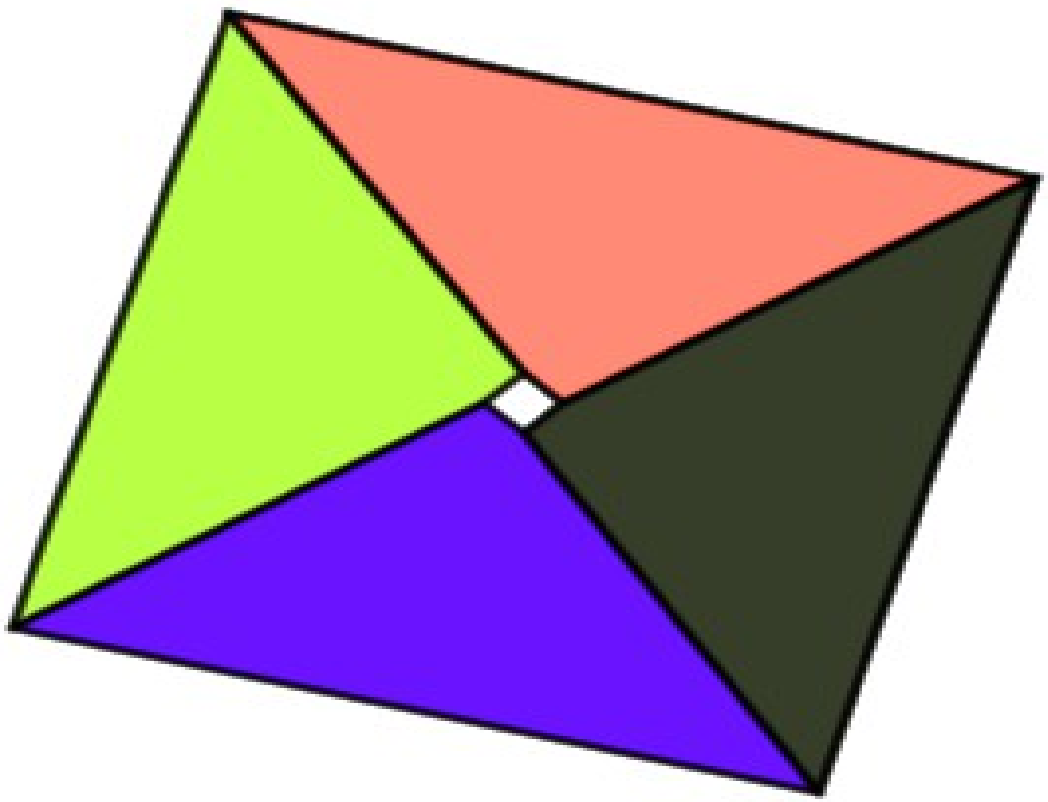


Illustration 7

```
int numPoints = int(map(62, 0, width, 6, 60));
```

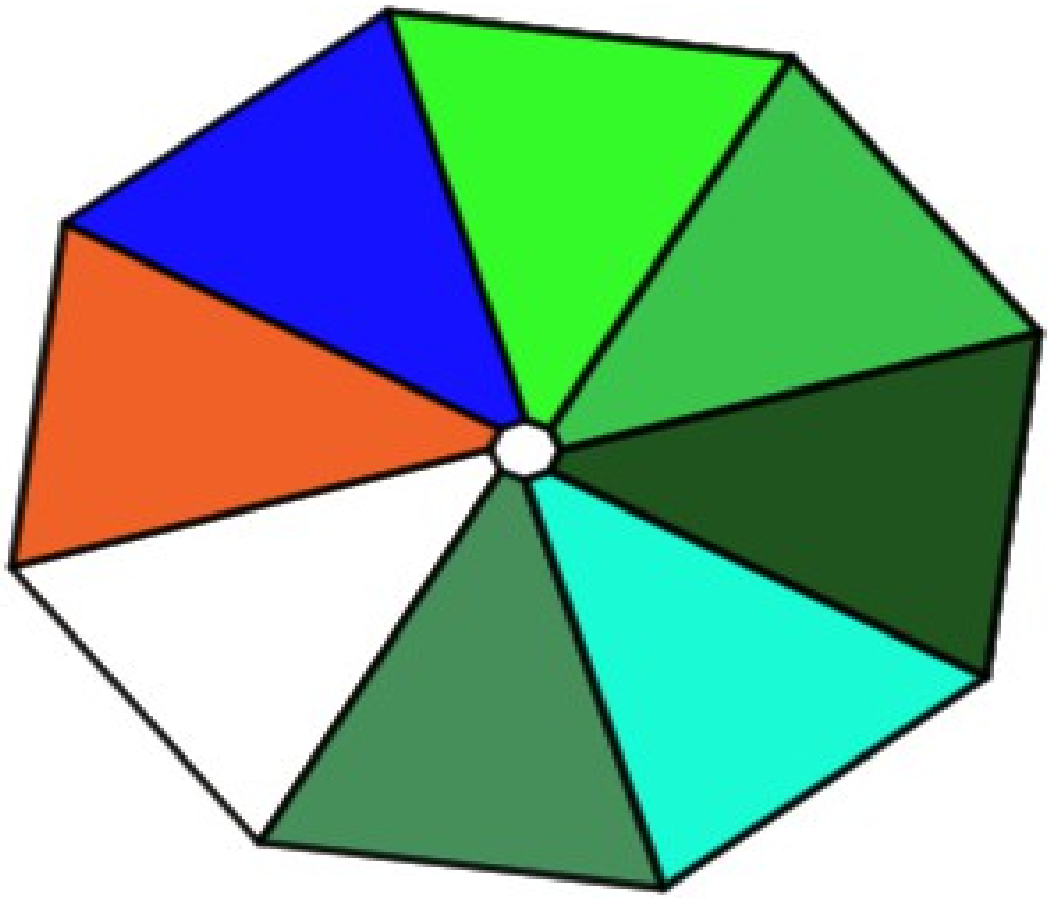


Illustration 8

```
int numPoints = int(map(435, 0, width, 6, 60));
```

Illustration 9

Shape 6

```
 float outsideRadius = 150;
float insideRadius = 10;
 float angleStep = 480.0/numPoints;
beginShape(PShape.QUAD_STRIP);
  for (int i = 0; i <= numPoints; i++) {
    float px = x + sin(radians(angle)) * insideRadius;
    float py = y + cos(radians(angle)) * insideRadius;
    angle += (angleStep);
    fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
    vertex(px, py);
    px = x + sin(radians(angle)) * outsideRadius;
    py = y + cos(radians(angle)) * outsideRadius;
    fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
    vertex(px, py);
    angle += (2*angleStep);
  }
  endShape();
}
```

```
int numPoints = int(map(47, 0, width, 6, 60));
```

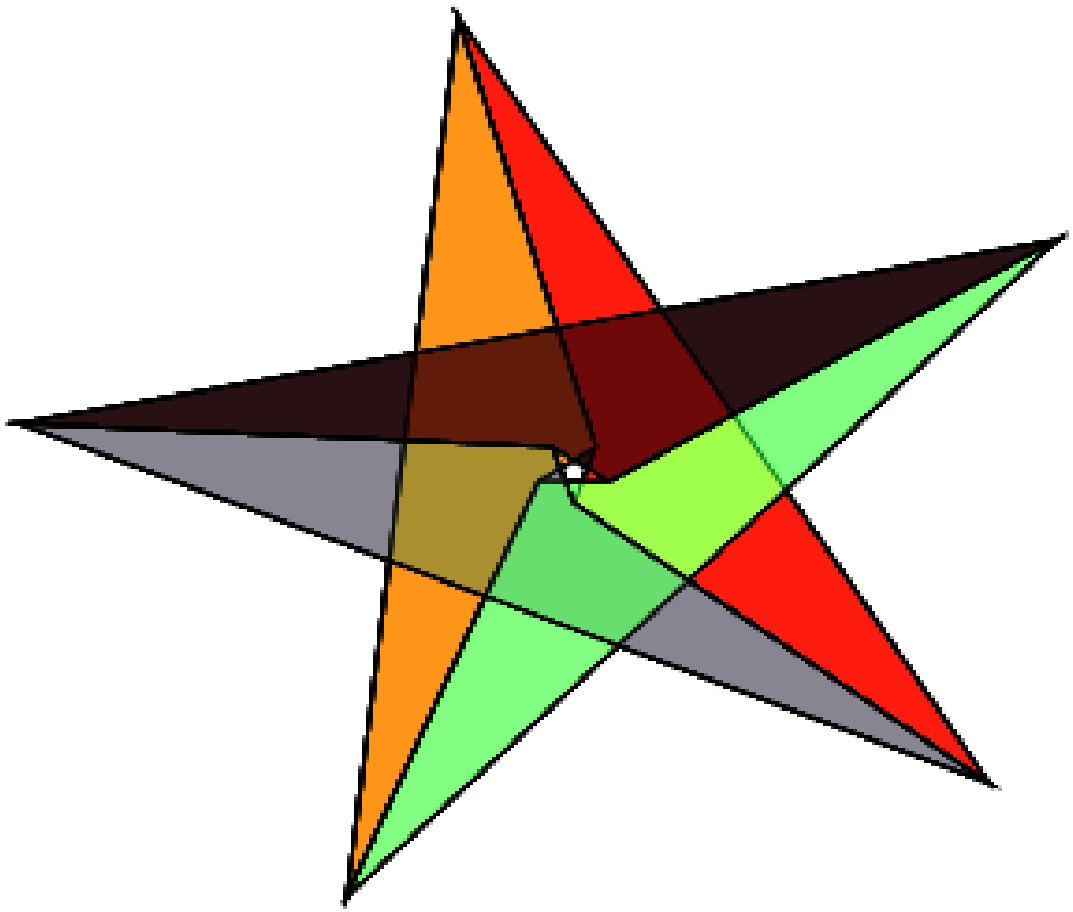


Illustration 10

```
int numPoints = int(map(35, 0, width, 6, 60));
```

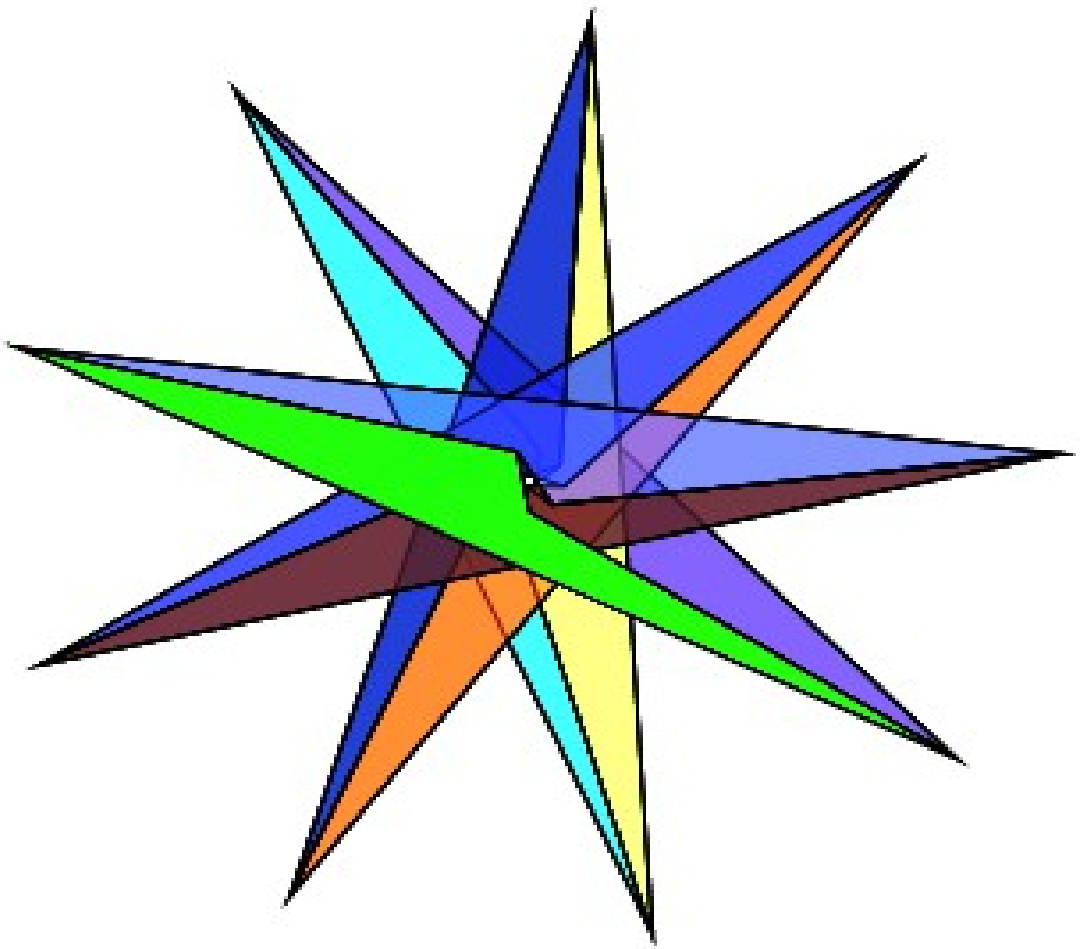


Illustration 11

```
int numPoints = int(map(135, 0, width, 6, 60));
```

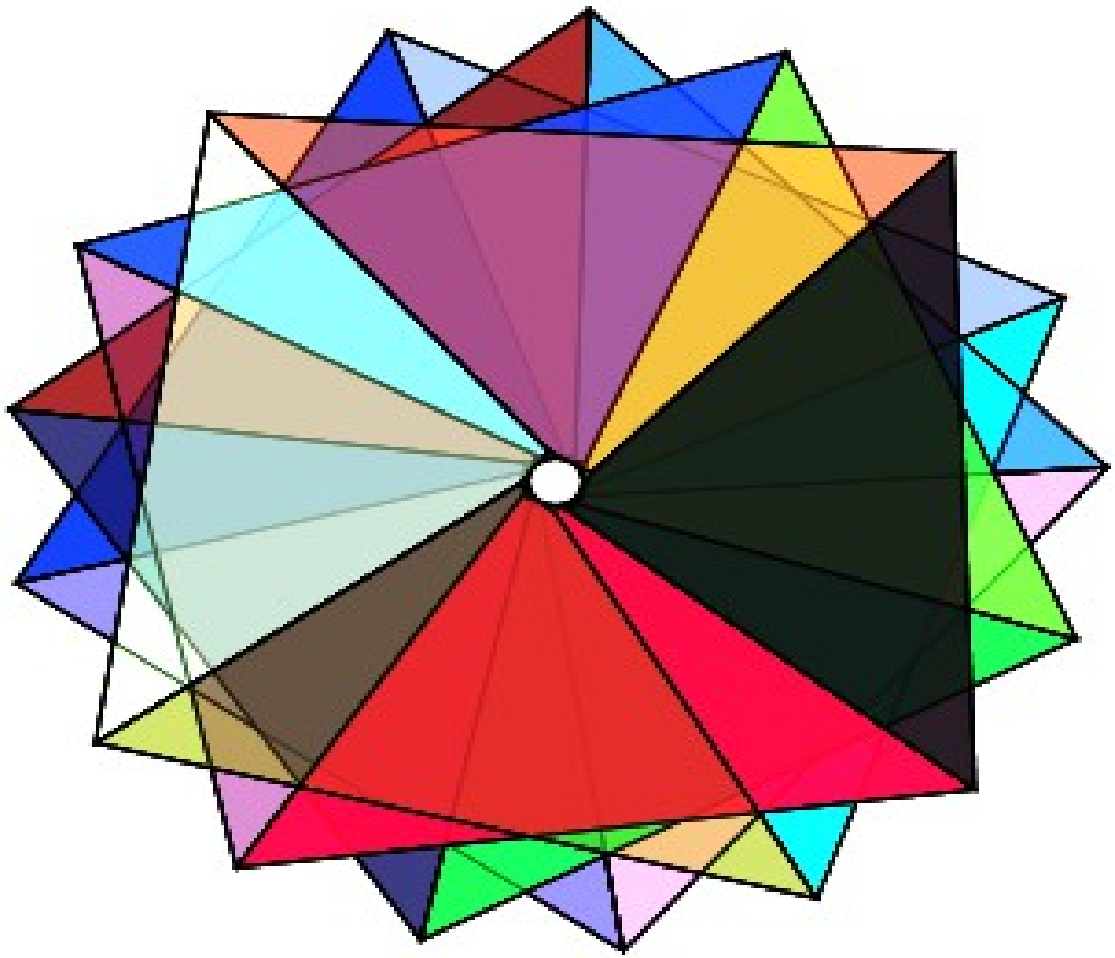


Illustration 12

```
int numPoints = int(map(254, 0, width, 6, 60));
```

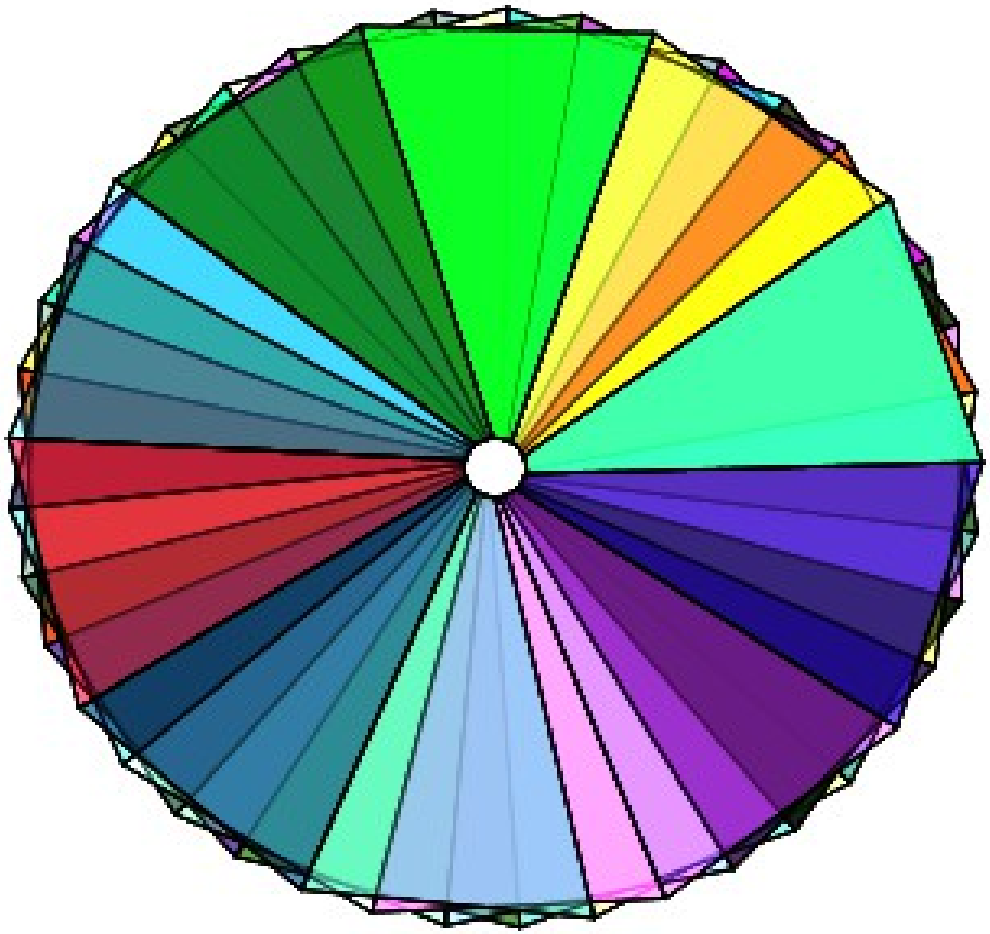


Illustration 13

Shape 7

```
 float outsideRadius = 150;
float insideRadius = 100;
float angleStep = 360.0/numPoints;
beginShape(PShape.QUAD_STRIP);
  for (int i = 0; i <= numPoints; i++) {
    float px = x + sin(radians(angle)) * insideRadius;
    float py = y + cos(radians(angle)) * insideRadius;
    angle += (angleStep);
    fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
    vertex(px, py);
    px = px*0.6 + sin(radians(angle)) * outsideRadius;
    py = py*0.6 + cos(radians(angle)) * outsideRadius;
    fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
    vertex(px, py);
    angle += (angleStep);
  }
  endShape();
```

```
int numPoints = int(map(455, 0, width, 6, 60));
```

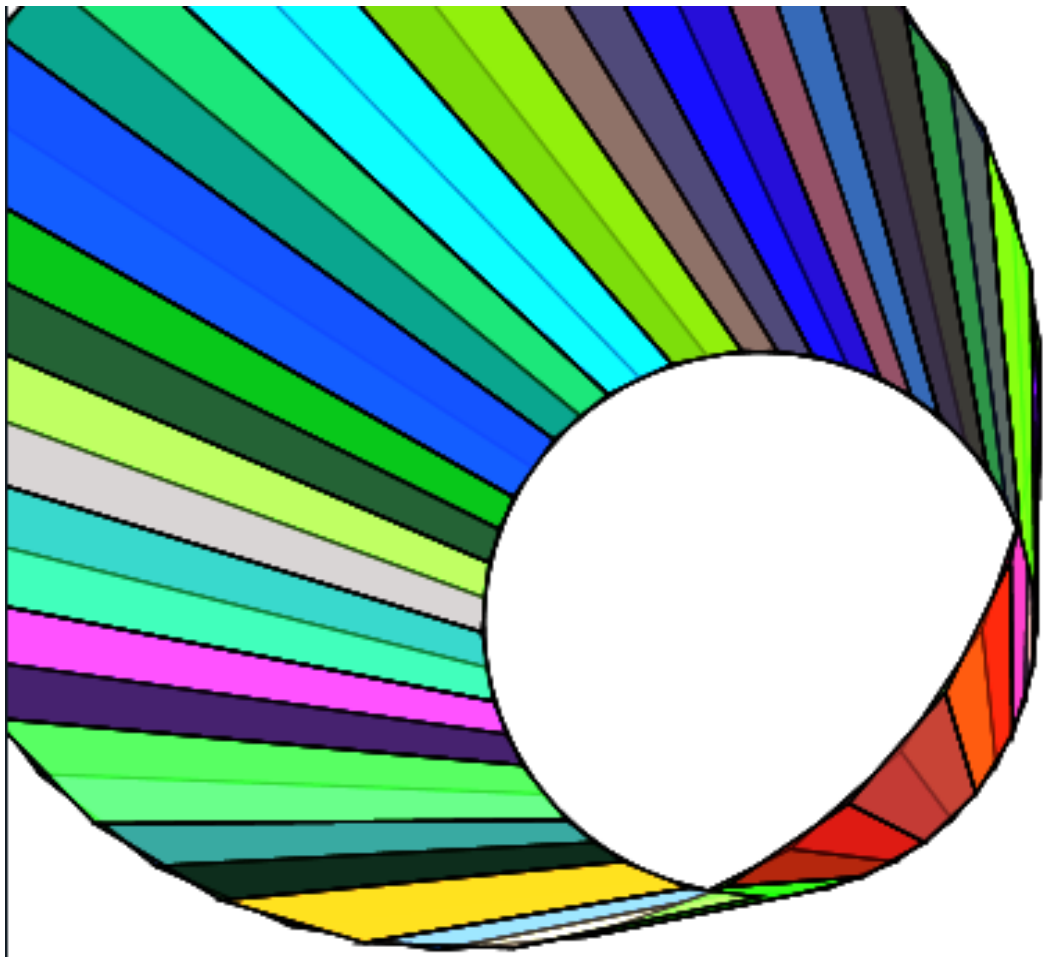


Illustration 14

Shape 8

```
beginShape(PShape.TRIANGLE_STRIP);
 for (int i = 0; i <= numPoints; i++) {
  float px = x + sin(radians(angle)) * insideRadius;
  float py = y + cos(radians(angle)) * insideRadius;
  angle += (angleStep);
  fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
  vertex(px, py);
  px = x + sin(radians(angle)) * outsideRadius;
  py = y + cos(radians(angle)) * insideRadius;
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
  angle += (2*angleStep);
  px = x + sin(radians(angle)) * outsideRadius;
  py = y + cos(radians(angle)) * insideRadius;
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
  angle += (2*angleStep);
  px = px + sin(radians(angle)) * outsideRadius;
  py = py + cos(radians(angle)) * outsideRadius;
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
  angle += (2*angleStep);
 }
 endShape();
```

```
int numPoints = int(map(501, 0, width, 6, 60));
```

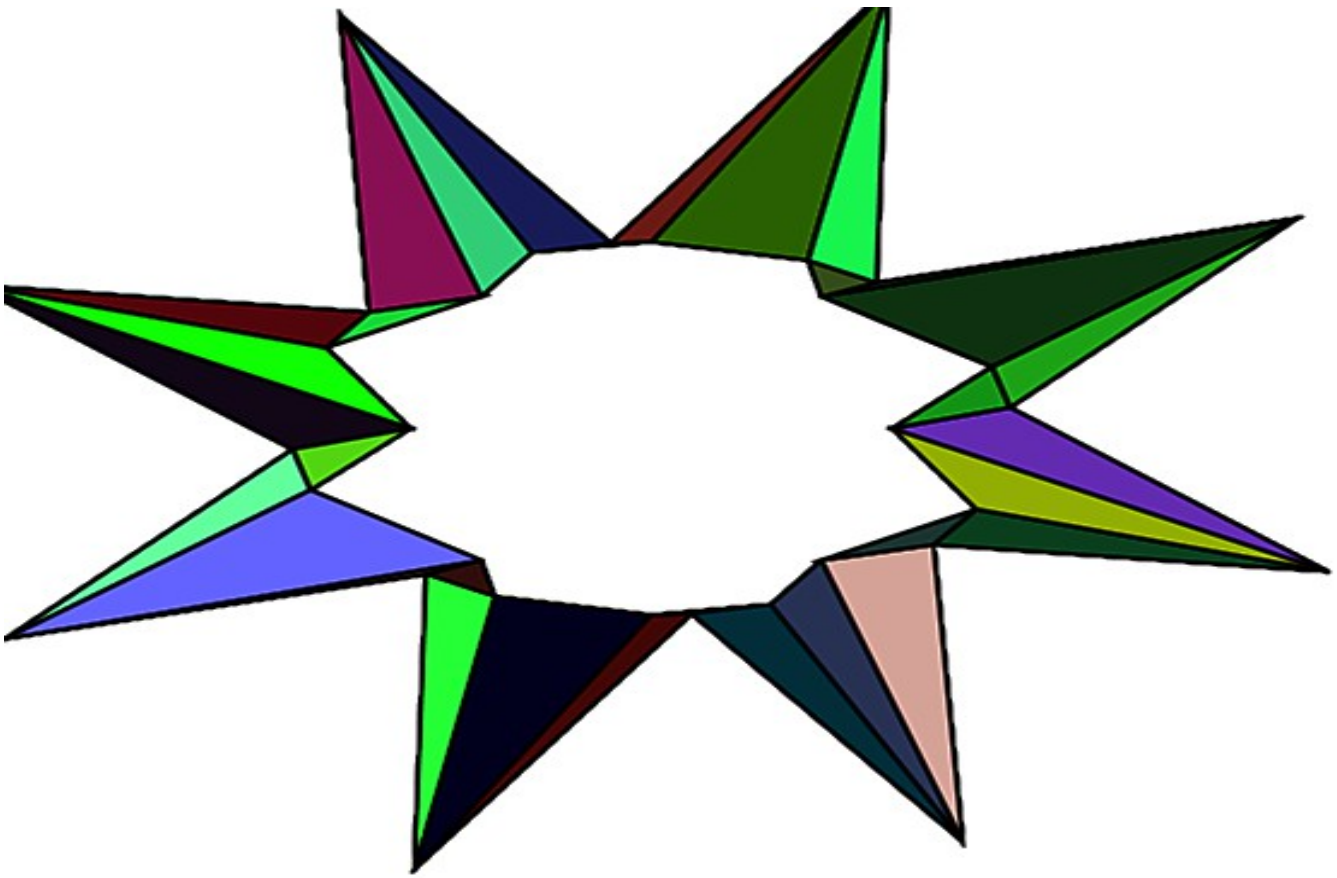


Illustration 15

```
int numPoints = int(map(223, 0, width, 6, 60));
```

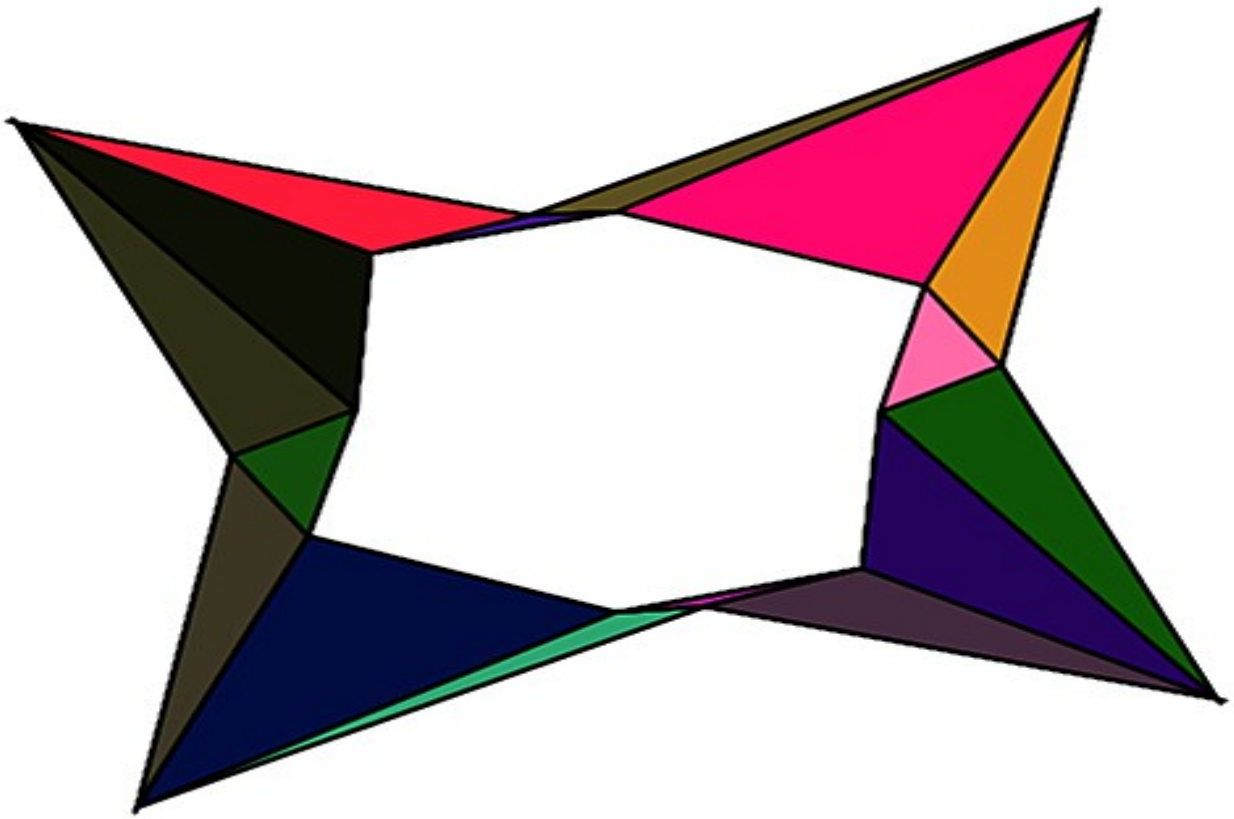


Illustration 16

```
int numPoints = int(map(473, 0, width, 6, 60));
```

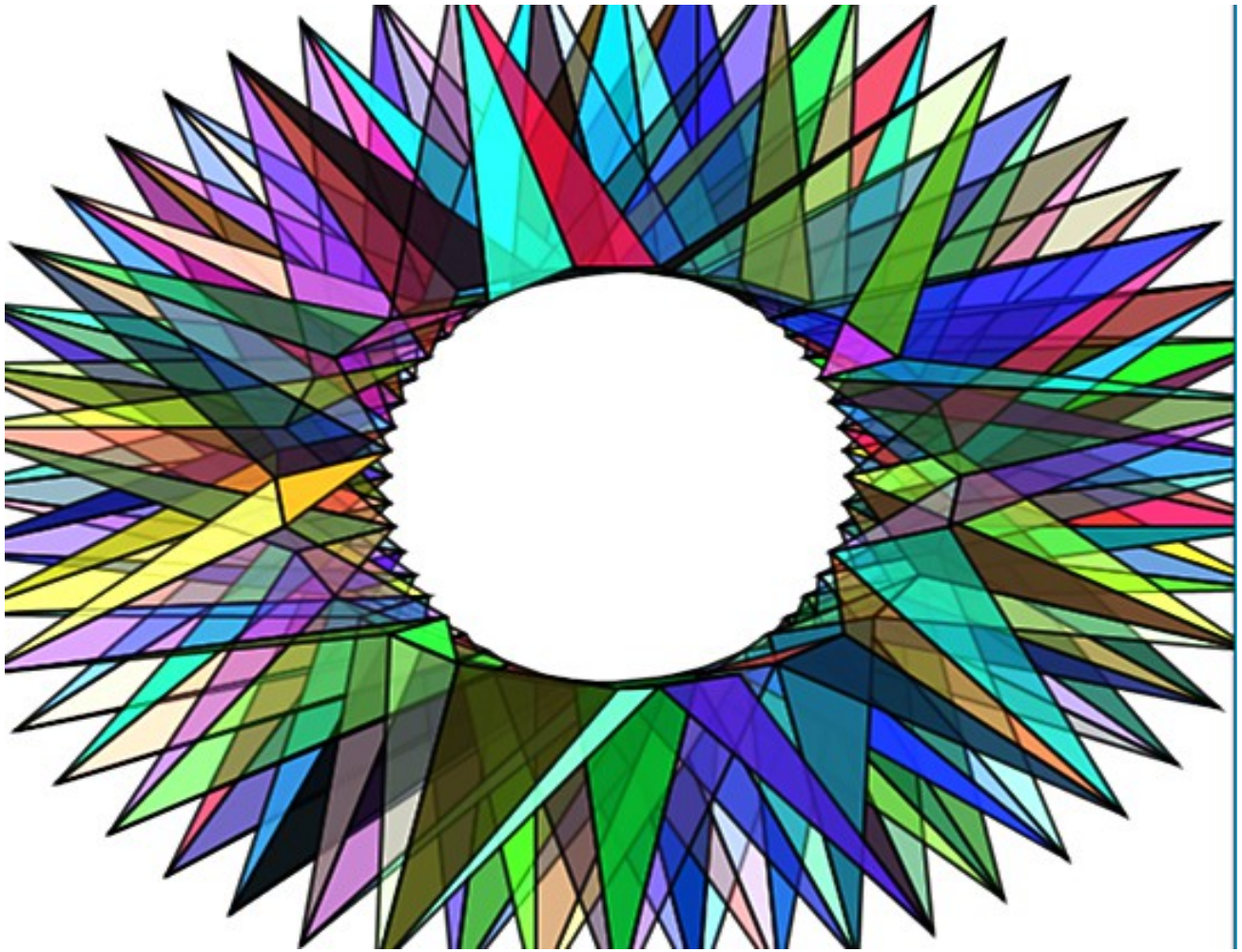


Illustration 17

Shape 9

```
beginShape(PShape.TRIANGLE_STRIP);
 for (int i = 0; i <= numPoints*10; i++) {
   float px = x + sin(radians(angle)) * outsideRadius;
   float py = y + cos(radians(angle)) * insideRadius;
   angle += (angleStep);
   fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
   vertex(px, py);
   px = x + sin(radians(angle)) * outsideRadius;
   py = y + cos(radians(angle)) * insideRadius;
   fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
   vertex(px, py);
   angle += (4*angleStep);
 }
 endShape();
```

```
int numPoints = int(map(6, 0, width, 6, 60));
```

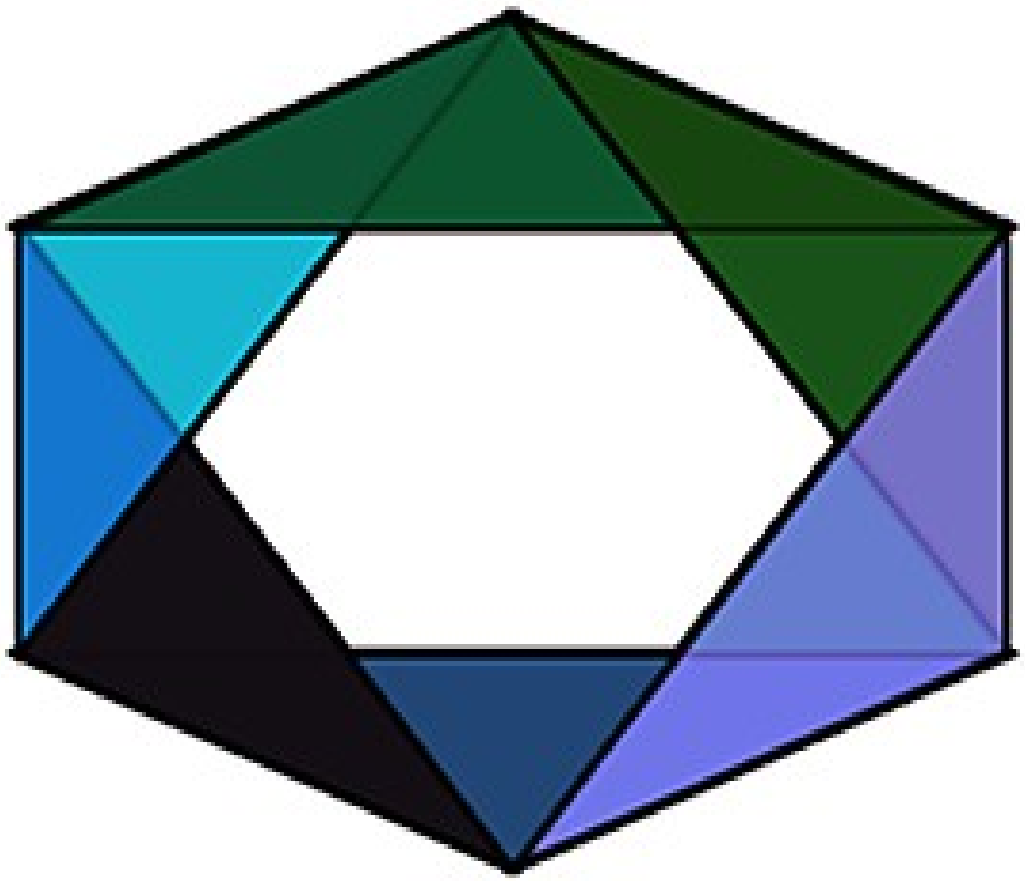


Illustration 18

```
int numPoints = int(map(6, 0, width, 6, 60));
```

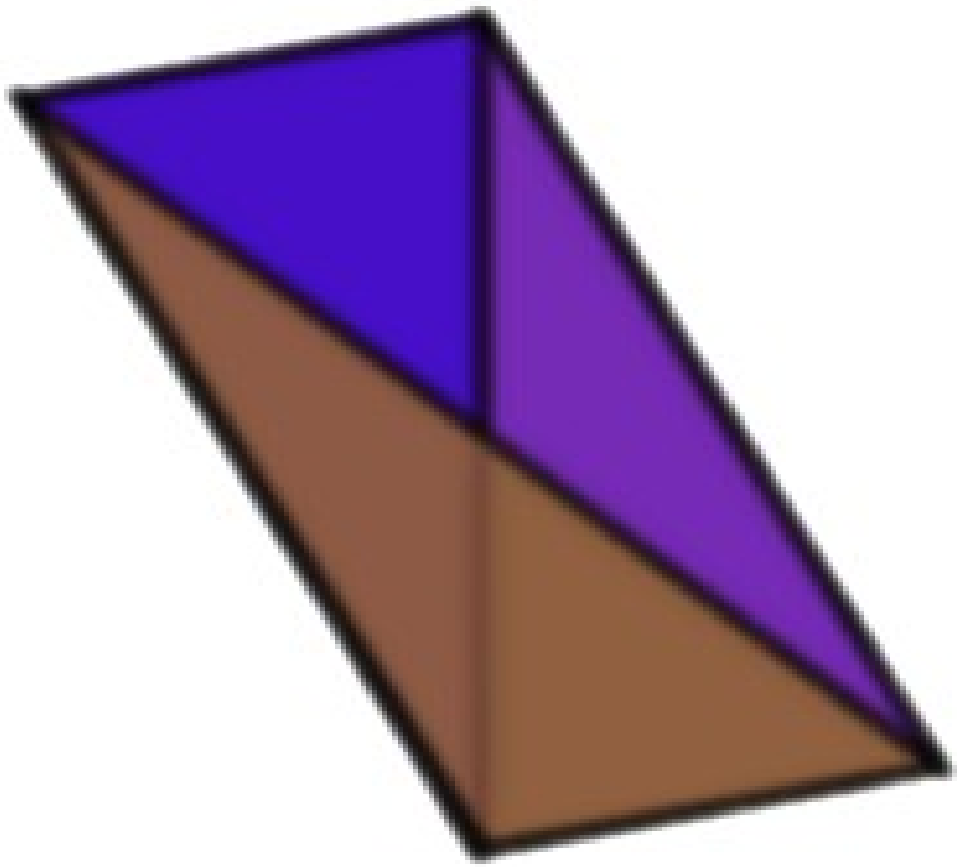


Illustration 19

```
int numPoints = int(map(230, 0, width, 6, 60));
```

Illustration 20

```
int numPoints = int(map(247, 0, width, 6, 60));
```

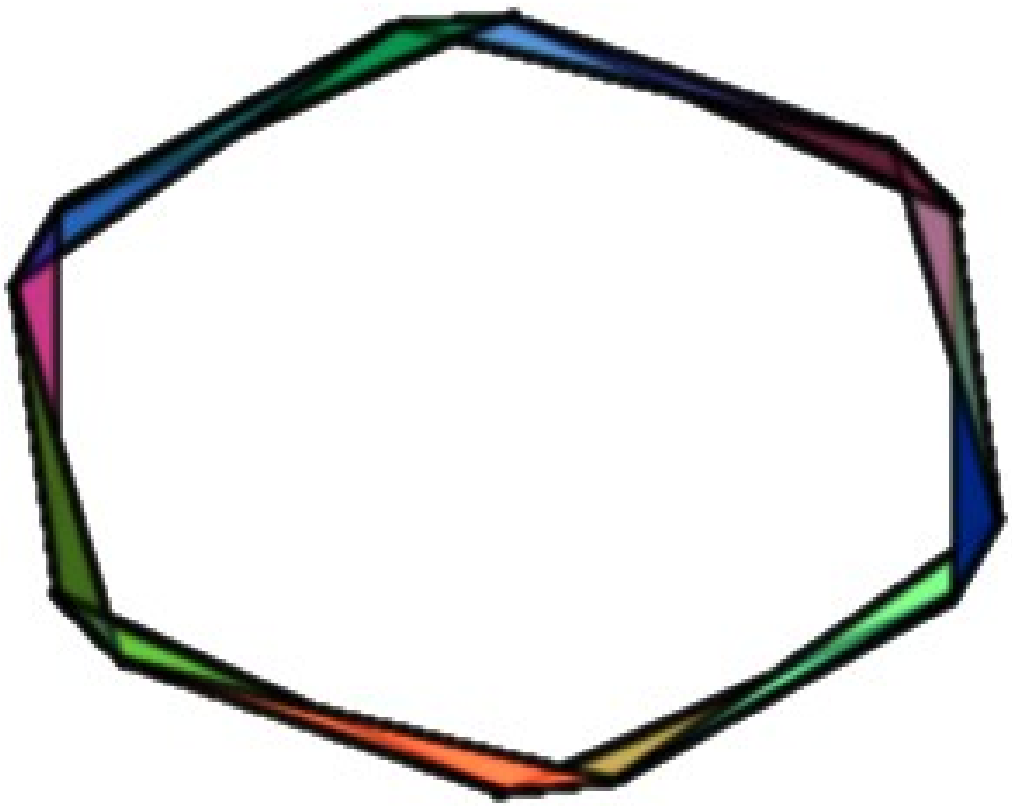


Illustration 21

```
int numPoints = int(map(415, 0, width, 6, 60));
```

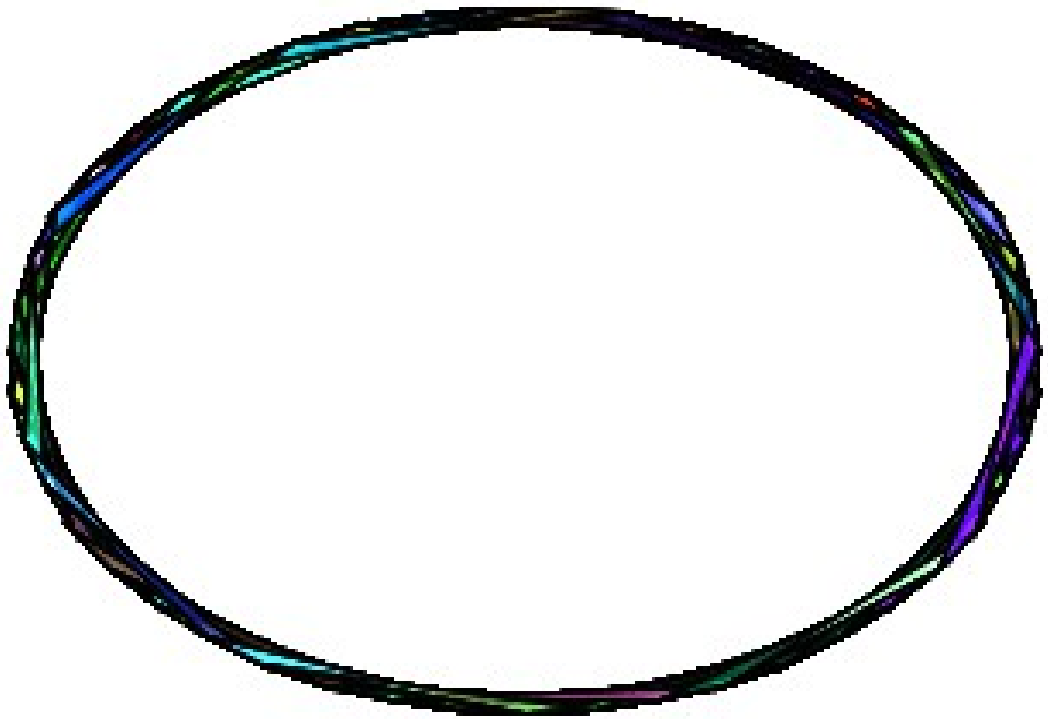


Illustration 22

```
int numPoints = int(map(56, 0, width, 6, 60));
```

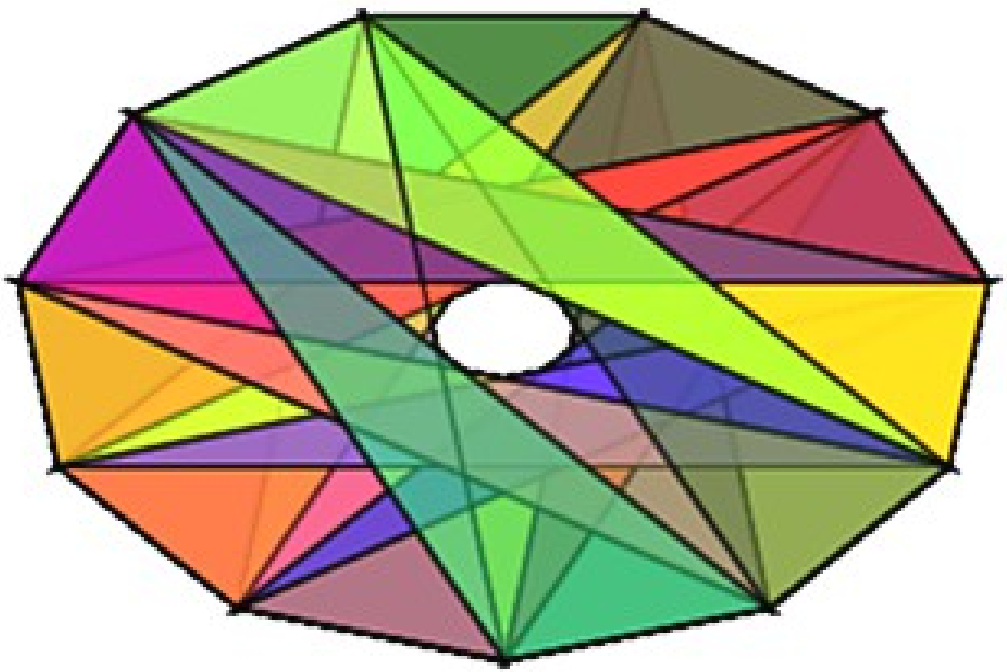


Illustration 23

Shape 10

```
beginShape(PShape.TRIANGLE_STRIP);
for (int i = 0; i <= numPoints; i++) {
  float px = x + sin(radians(angle)) * insideRadius;
  float py = y + cos(radians(angle)) * insideRadius;
  angle += (angleStep);
  fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
  vertex(px, py);
  px = x + cos(radians(angle)) * outsideRadius;
  py = y + sin(radians(angle)) * outsideRadius;
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
  angle += (angleStep);
}
endShape();
```

```
int numPoints = int(map(490, 0, width, 6, 60));
```

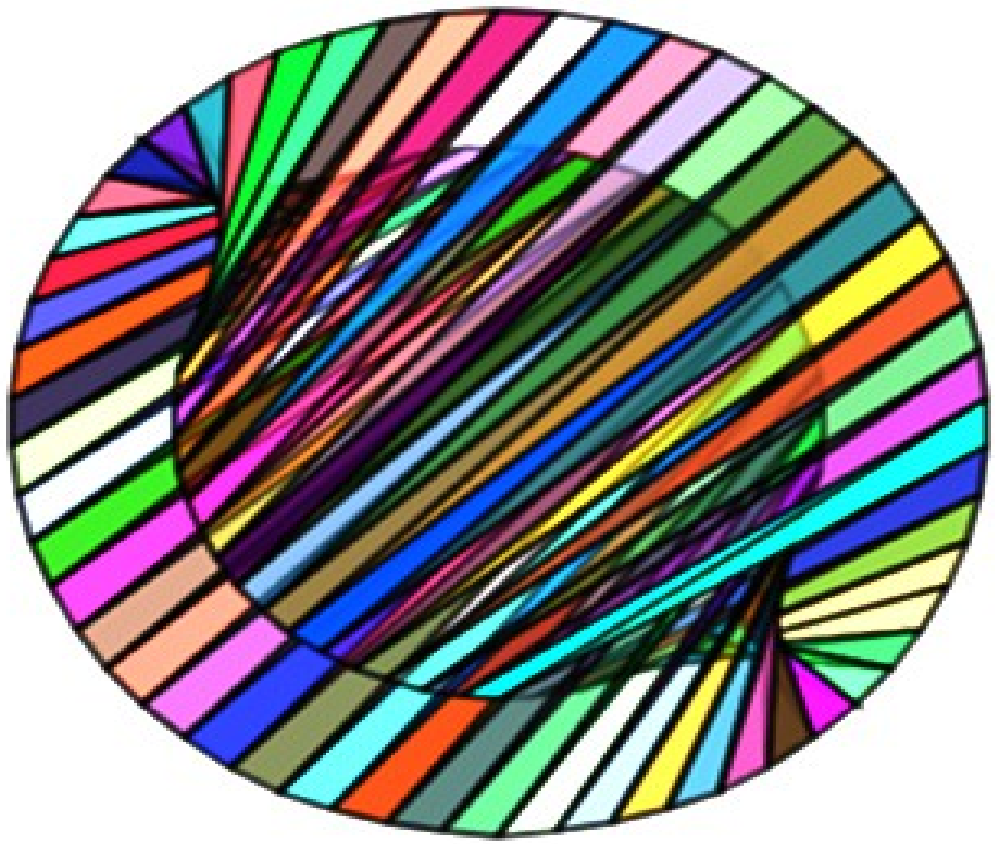


Illustration 24

Shape 11

```
beginShape(PShape.QUAD_STRIP);
 for (int i = 0; i <= numPoints; i++) {
   float px = x + sin(radians(angle)) * insideRadius;
   float py = y + cos(radians(angle)) * insideRadius;
   angle += (angleStep);
   fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
   vertex(px, py);
   px = x + cos(radians(angle*2)) * outsideRadius;
   py = y + sin(radians(angle*2)) * outsideRadius;
   fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
   vertex(px, py);
   angle += (angleStep);
 }
 endShape();
```

```
int numPoints = int(map(455, 0, width, 6, 60));
```

Illustration 25

Shape 12

```
beginShape(PShape.QUAD_STRIP);
 for (int i = 0; i <= numPoints; i++) {
  float px = x + cos(radians(angle)) * insideRadius;
  float py = y + sin(radians(angle)) * insideRadius;
  angle += (angleStep);
  fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
  vertex(px, py);
  px = x + sin(radians(log(angle))) * outsideRadius;
  py = y + sin(radians(log(angle))) * outsideRadius;
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
  angle += (2*angleStep);
  px = x + sin(radians(log(angle))) * outsideRadius;
  py = y + sin(radians(log(angle))) * outsideRadius;
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
  angle += (2*angleStep);px = x + sin(radians(log(angle))) * outsideRadius;
  py = y + sin(radians(log(angle))) * outsideRadius;
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
  angle += (2*angleStep);
 }
 endShape();
```

```
int numPoints = int(map(55, 0, width, 6, 60));
```

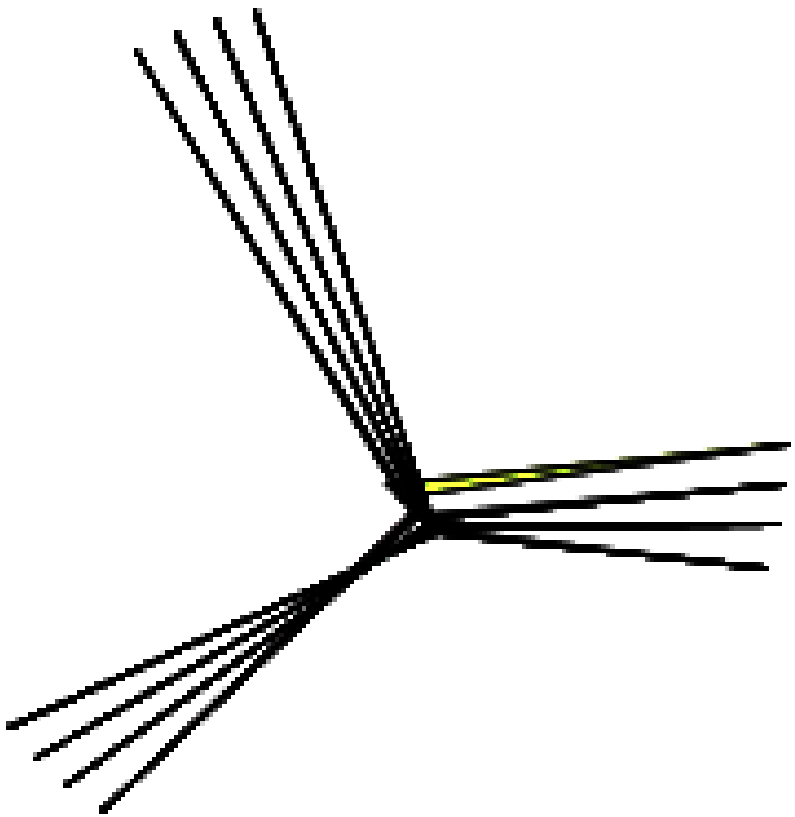


Illustration 26

```
int numPoints = int(map(380, 0, width, 6, 60));
```

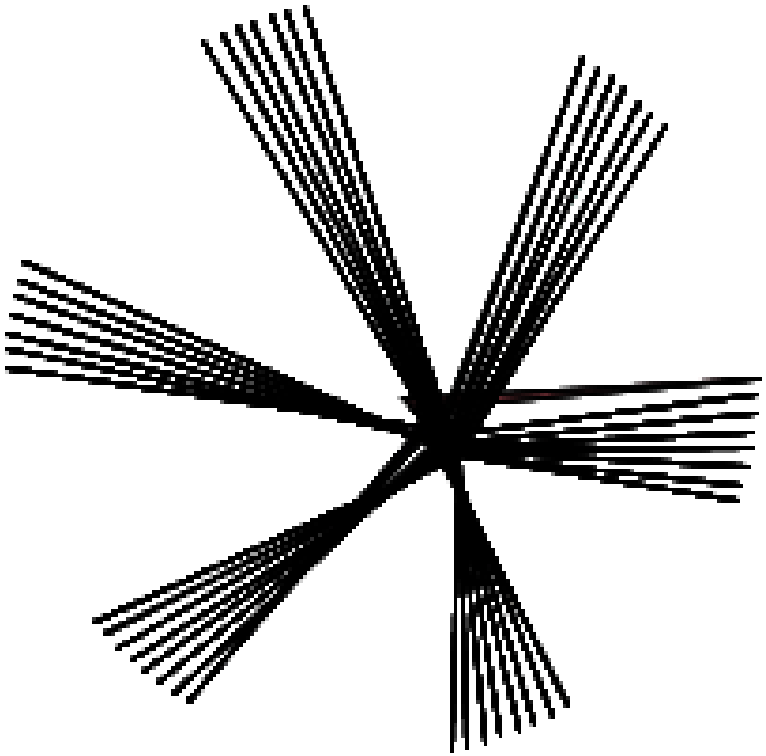


Illustration 27

```
int numPoints = int(map(450, 0, width, 6, 60));
```

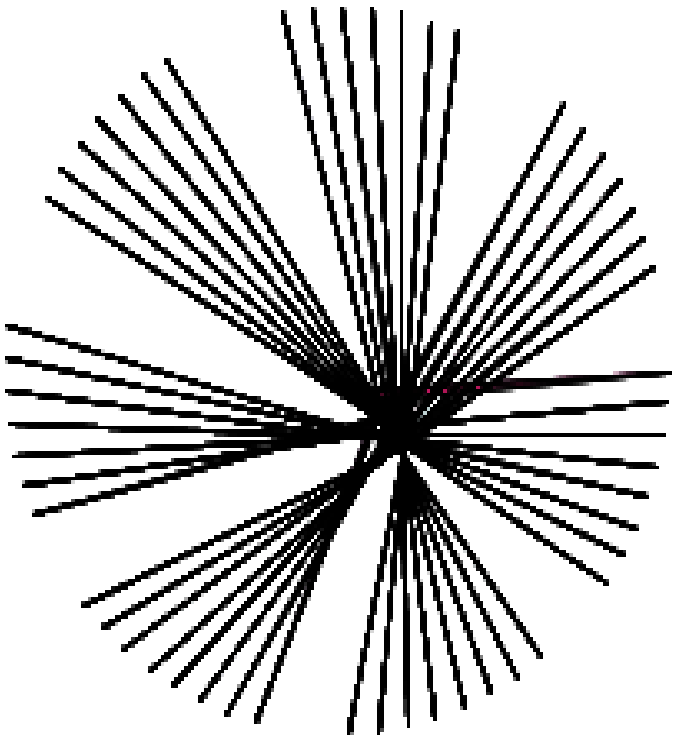


Illustration 28

```
int numPoints = int(map(490, 0, width, 6, 60));
```

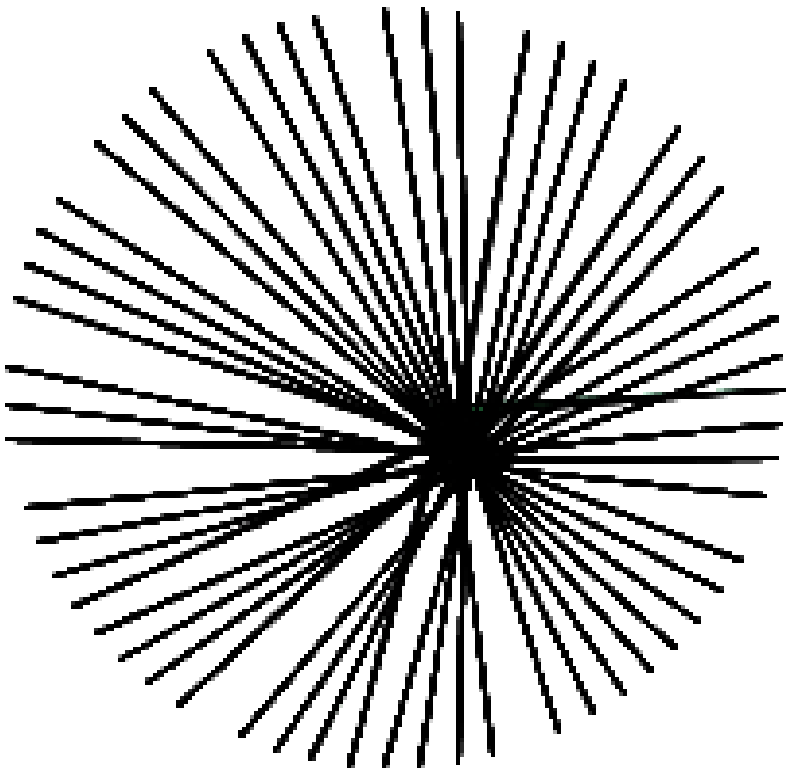


Illustration 29

```
 beginShape(PShape.TRIANGLE_STRIP);                                    Shape 13
 for (int i = 0; i <= numPoints; i++) {
   float px = x + cos(radians(angle)) * insideRadius;
   float py = y + sin(radians(angle)) * insideRadius;
   angle += (angleStep);
   fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
   vertex(px, py);
   angle += (angleStep);
   px = x + cos(radians(angle)) * insideRadius;
   py = y + sin(radians(angle)) * outsideRadius;
   fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
   vertex(px, py);
 }
 endShape();
}
```

```
int numPoints = int(map(7, 0, width, 6, 60));
float outsideRadius = 150;
float insideRadius = 100;
```

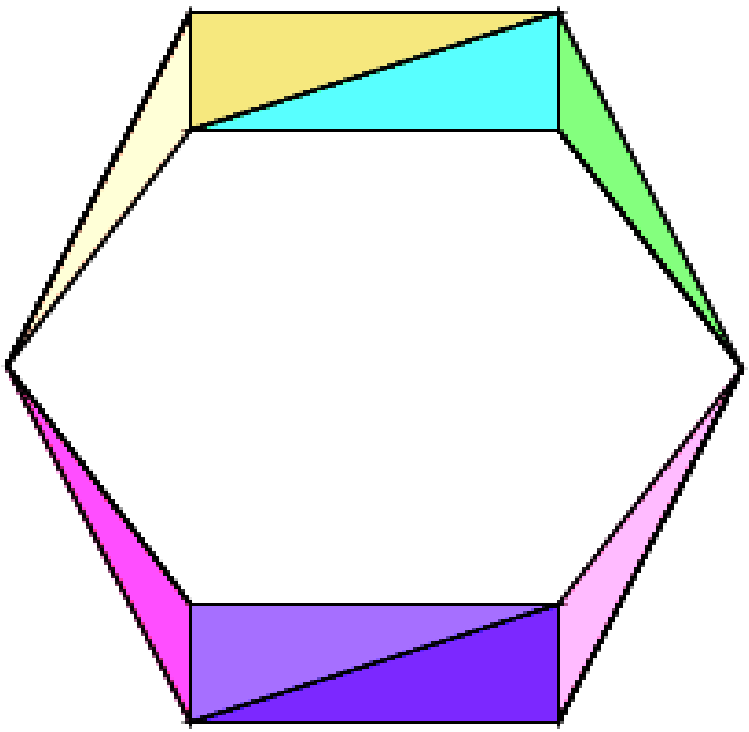


Illustration 30

```
int numPoints = int(map(440, 0, width, 6, 60));
float outsideRadius = 150;
float insideRadius = 130;
```

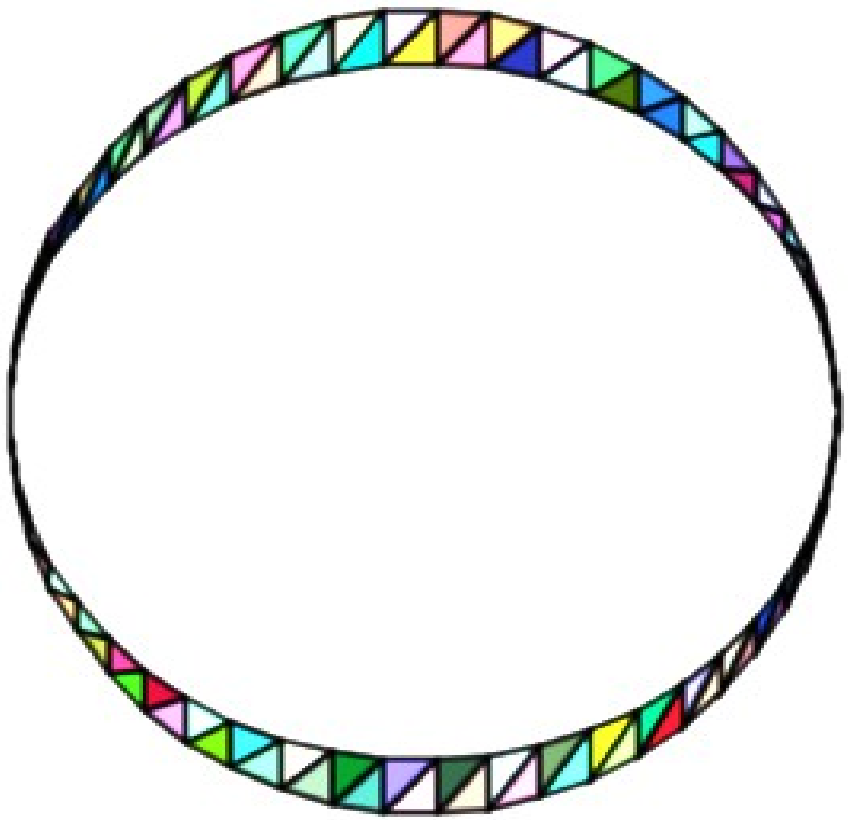


Illustration 31

Shape 14

```
 beginShape(PShape.TRIANGLE_STRIP);
  for (int i = 0; i <= numPoints; i++) {
    float px =x + cos(radians(angle)) * insideRadius;
    float py = y + sin(radians(angle)) * insideRadius;
    angle += (angleStep);
    fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
    vertex(px, py);
    px = x+cos(radians(angle)) * outsideRadius;
    py = y+sin(radians(angle)) * outsideRadius;
    angle += (angleStep);
    fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
    vertex(px, py);
    px = x+cos(radians(angle)) * outsideRadius;
   py = y+sin(radians(angle)) * outsideRadius;
   angle += (angleStep);
   fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
   vertex(px, py);
    px = x+cos(radians(angle)) * insideRadius;
   py = y+sin(radians(angle)) * insideRadius;
    angle += (angleStep);
    fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
   vertex(px, py); }
endShape();
```

```
int numPoints = int(map(680, 0, width, 6, 60));
```

Illustration 32

```
int numPoints = int(map(223, 0, width, 6, 60));
```

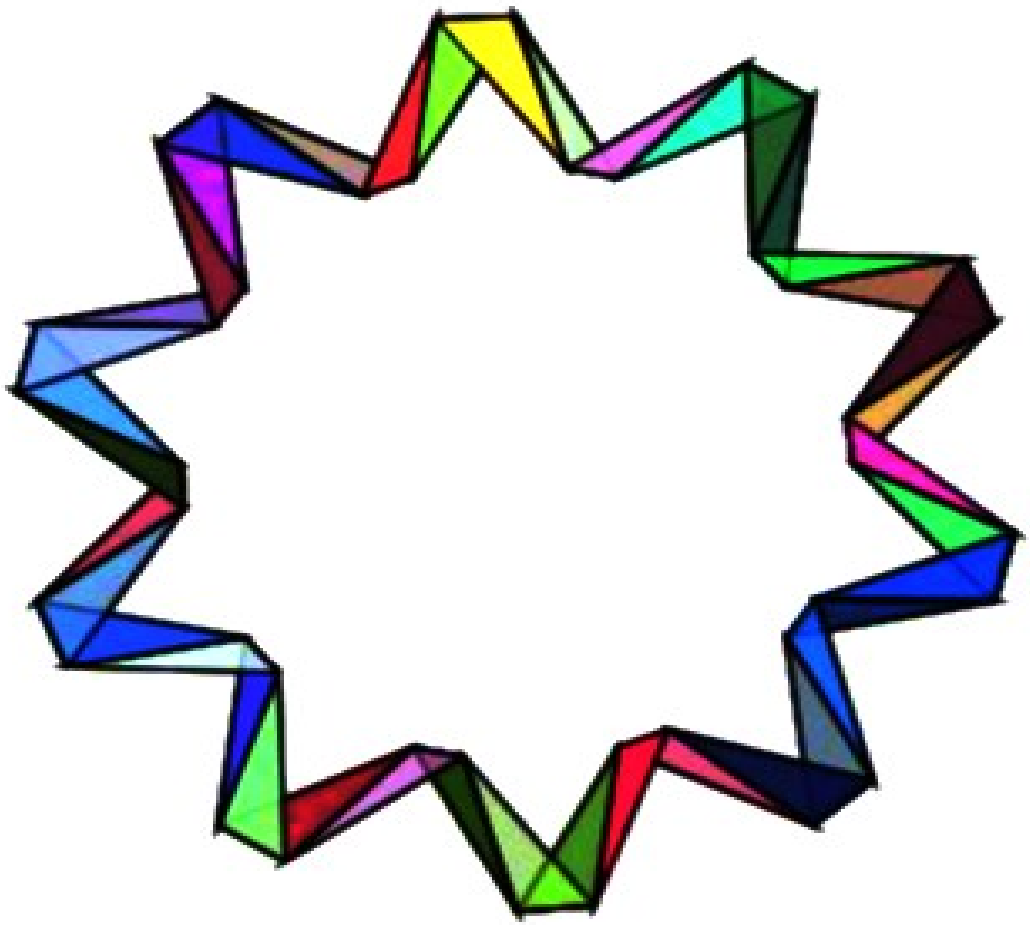


Illustration 33

```
int numPoints = int(map(46, 0, width, 6, 60));
```

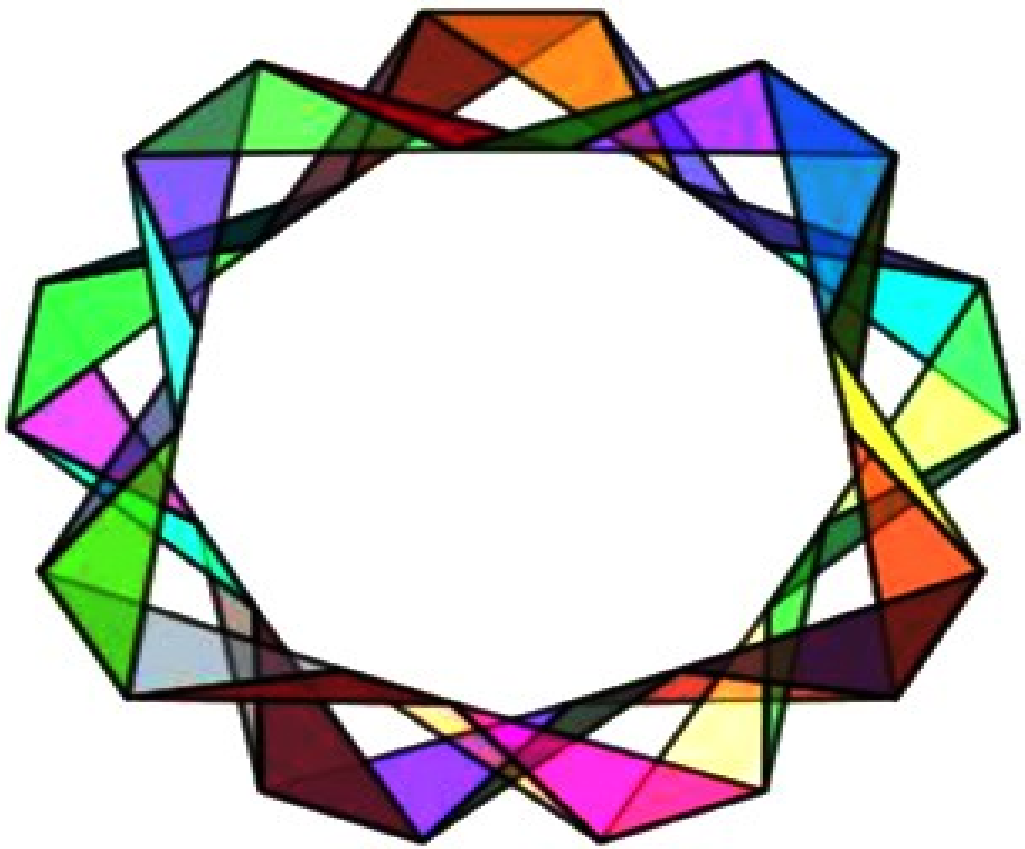


Illustration 34

```
int numPoints = int(map(26, 0, width, 6, 60));
```

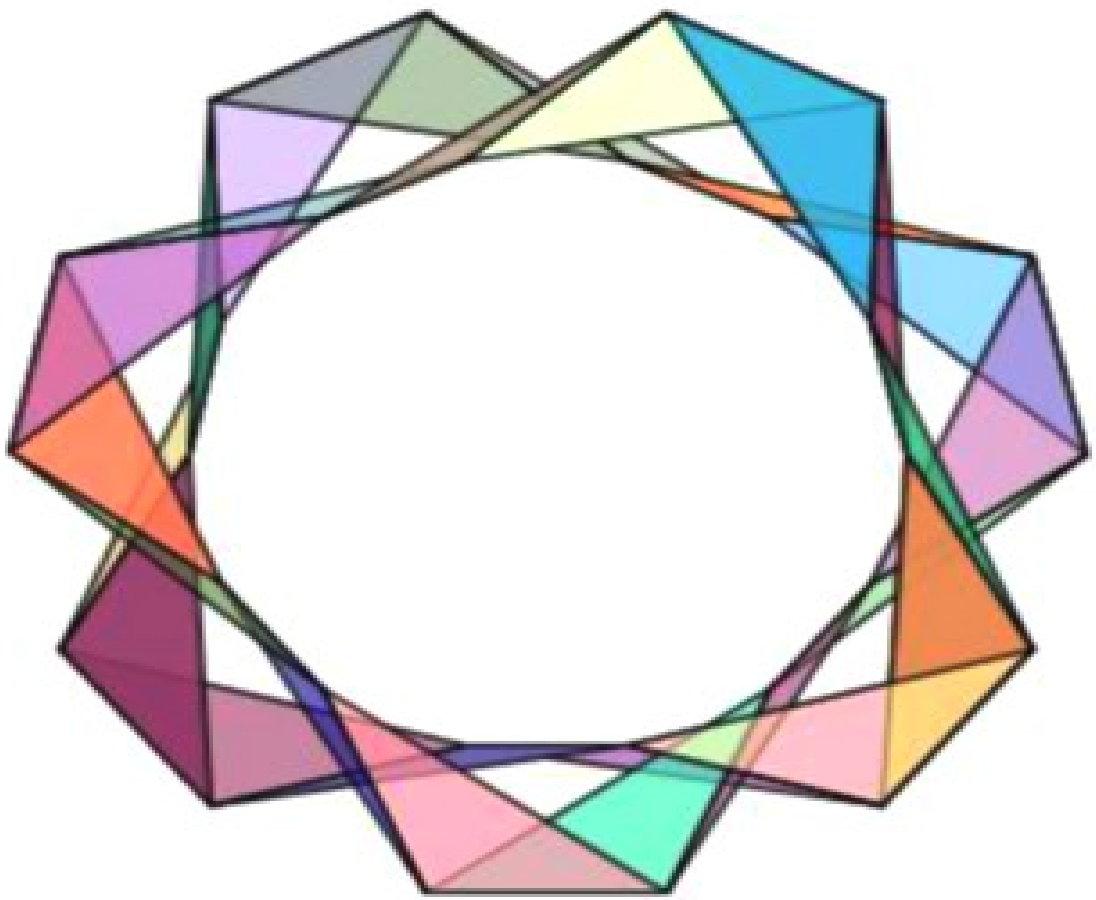


Illustration 35

```
int numPoints = int(map(6, 0, width, 6, 60));
```

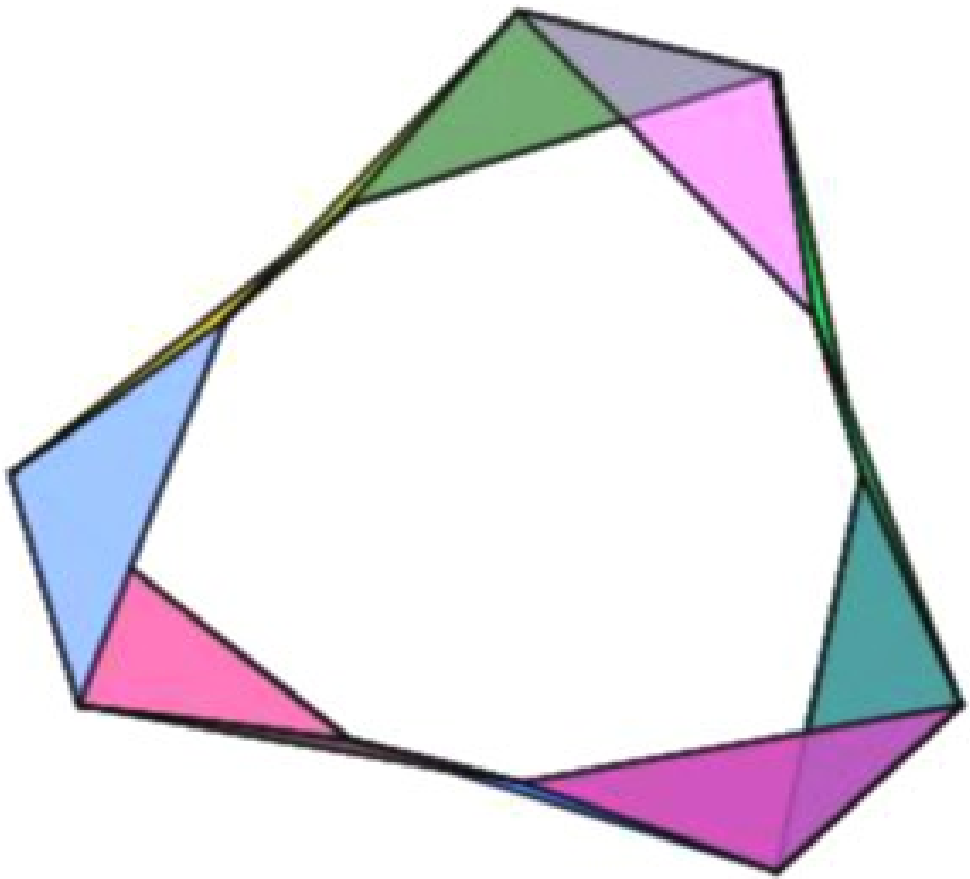


Illustration 36

```
int numPoints = int(map(93, 0, width, 6, 60));
```

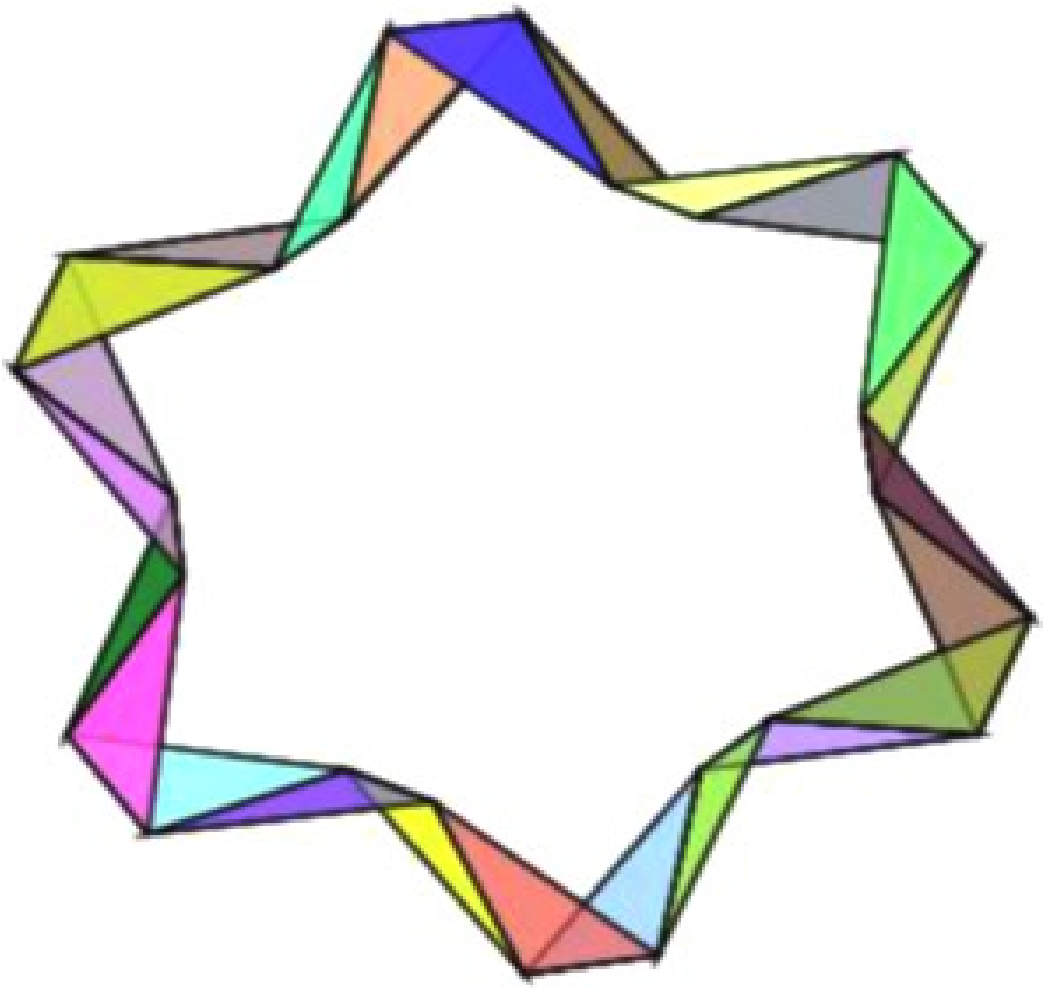


Illustration 37

Shape 15

```
beginShape(PShape.QUAD_STRIP);
 for (int i = 0; i <= numPoints; i++) {
   float px =x + cos(radians(angle)) * outsideRadius;
   float py = y + sin(radians(angle)) * outsideRadius;
   angle += (angleStep);
   fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
   vertex(px, py);
   px = log(px);
   py = log(py);
   angle1 += (angleStep);
   fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
   vertex(px, py);
 }
 endShape();
```

```
int numPoints = int(map(386, 0, width, 6, 60));
```

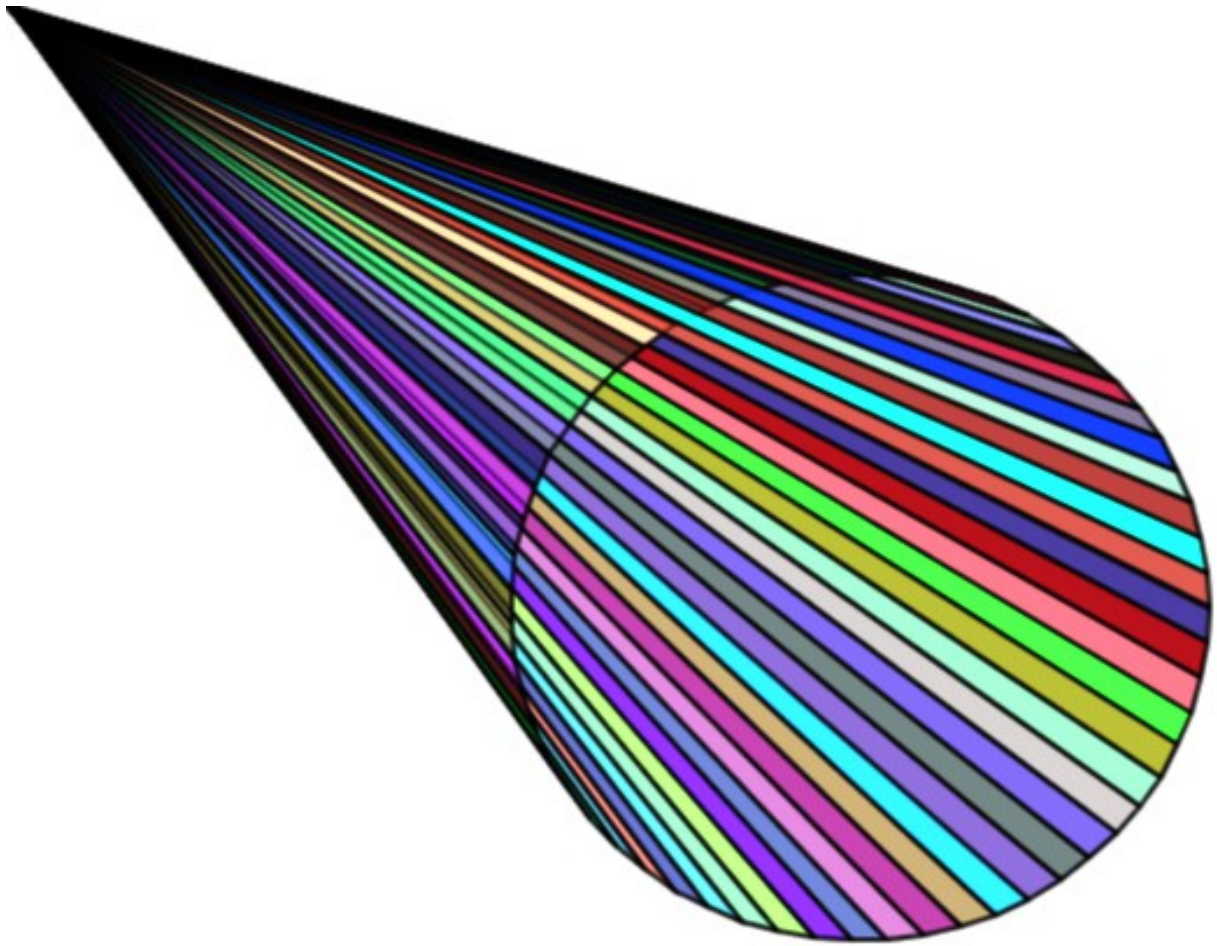


Illustration 38

```
float angleStep = 360.0/numPoints;                                    Shape 16
beginShape(PShape.TRIANGLE_STRIP);
for (int i = 0; i <= numPoints; i++) {
  float px =x + sin(radians(angle)) * outsideRadius;
  float py =y + cos(radians(angle)) * outsideRadius;
  angle += (angleStep);
  fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
  vertex(px, py);
  px = x + sin(radians(angle1-angle)) * insideRadius;
  py = y + cos(radians(angle1-angle)) * insideRadius;
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  angle1 += (angleStep);
  vertex(px, py);
}
endShape();
```

```
int numPoints = int(map(692, 0, width, 6, 60));
```

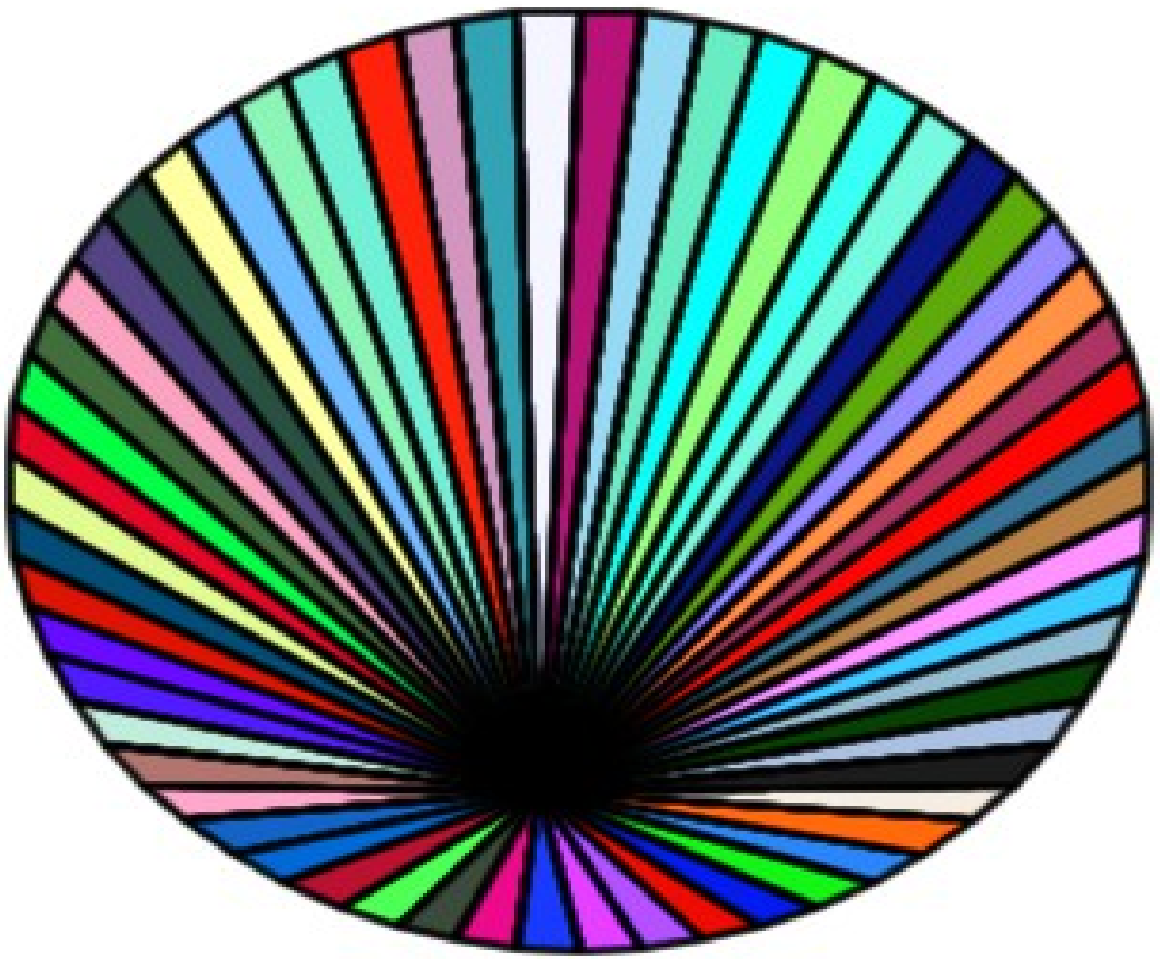


Illustration 39

```
int numPoints = int(map(63, 0, width, 6, 60));
```

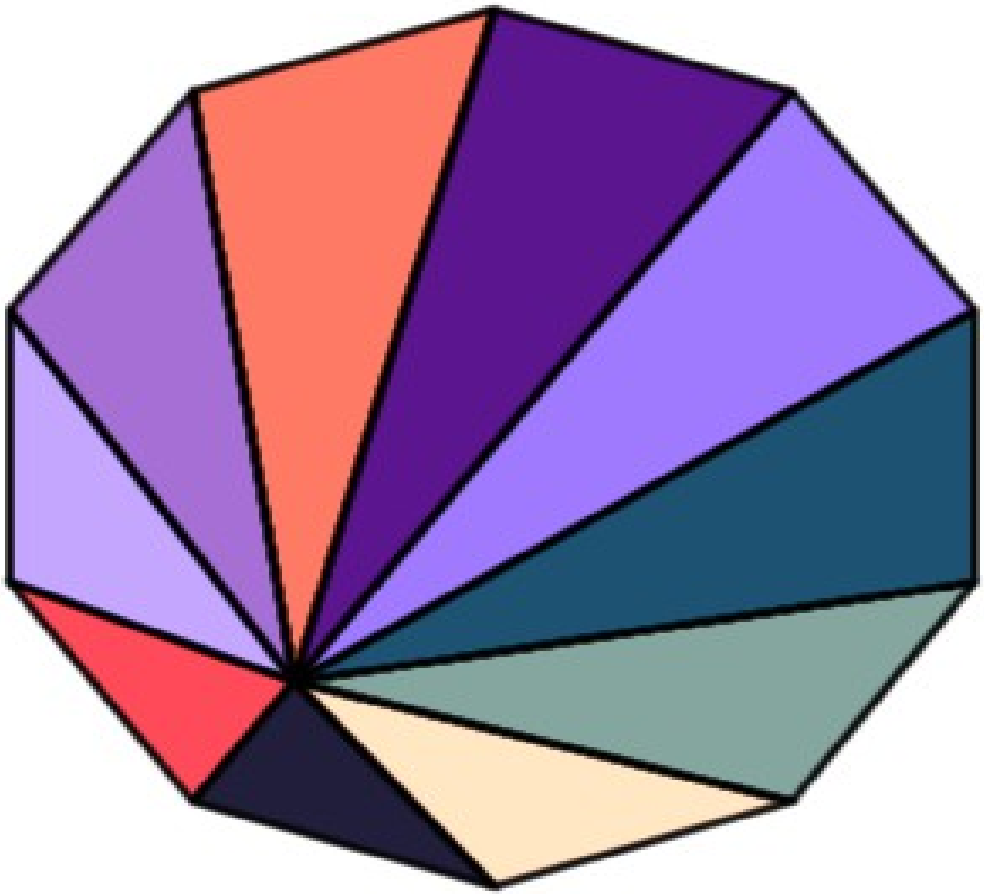


Illustration 40

Shape 17

```
beginShape(PShape.TRIANGLE_STRIP);
 for (int i = 0; i <= numPoints; i++) {
   float px =x + cos(radians(2*angle)) * insideRadius;
   float py = y + sin(radians(2*angle)) * insideRadius;
   angle += (angleStep);
   fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
   vertex(px, py);
   px = log(x) ;
   py = log(y);
   angle += (angleStep);
   fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
   vertex(px, py);
 }
 endShape();
```

```
int numPoints = int(map(64, 0, width, 6, 60));
```

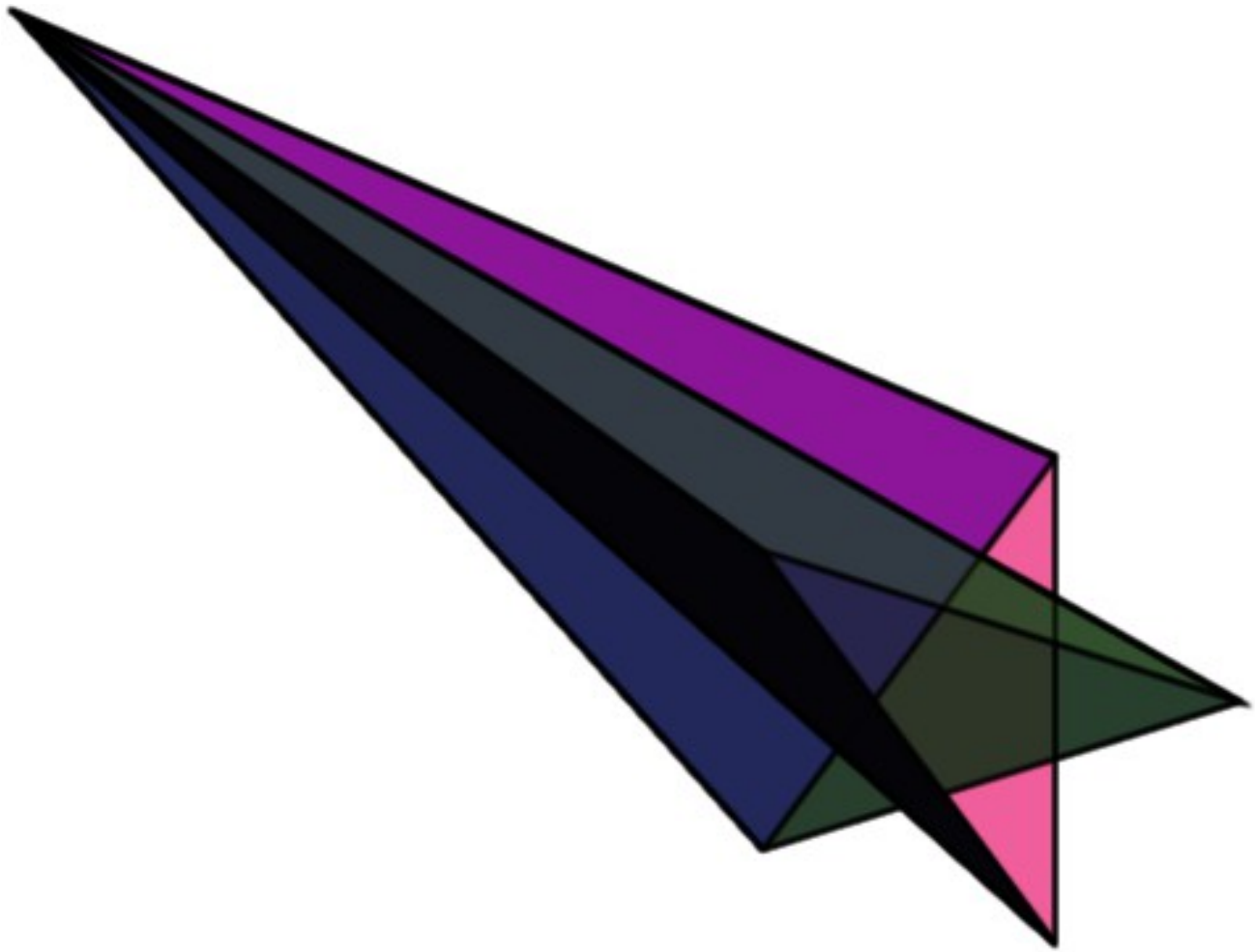


Illustration 41

```
int numPoints = int(map(200, 0, width, 6, 60));
```

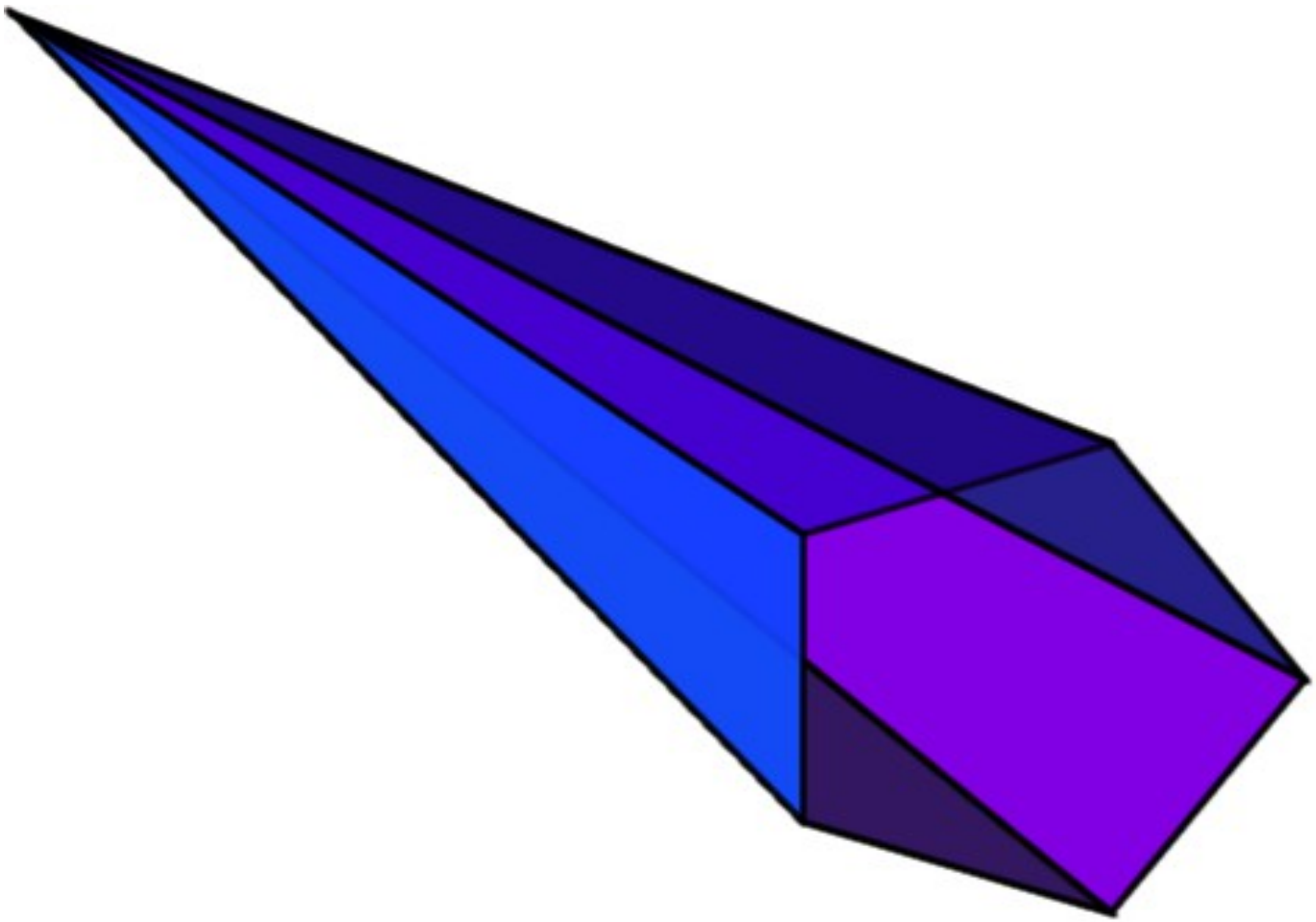


Illustration 42

```
int numPoints = int(map(166, 0, width, 6, 60));
```

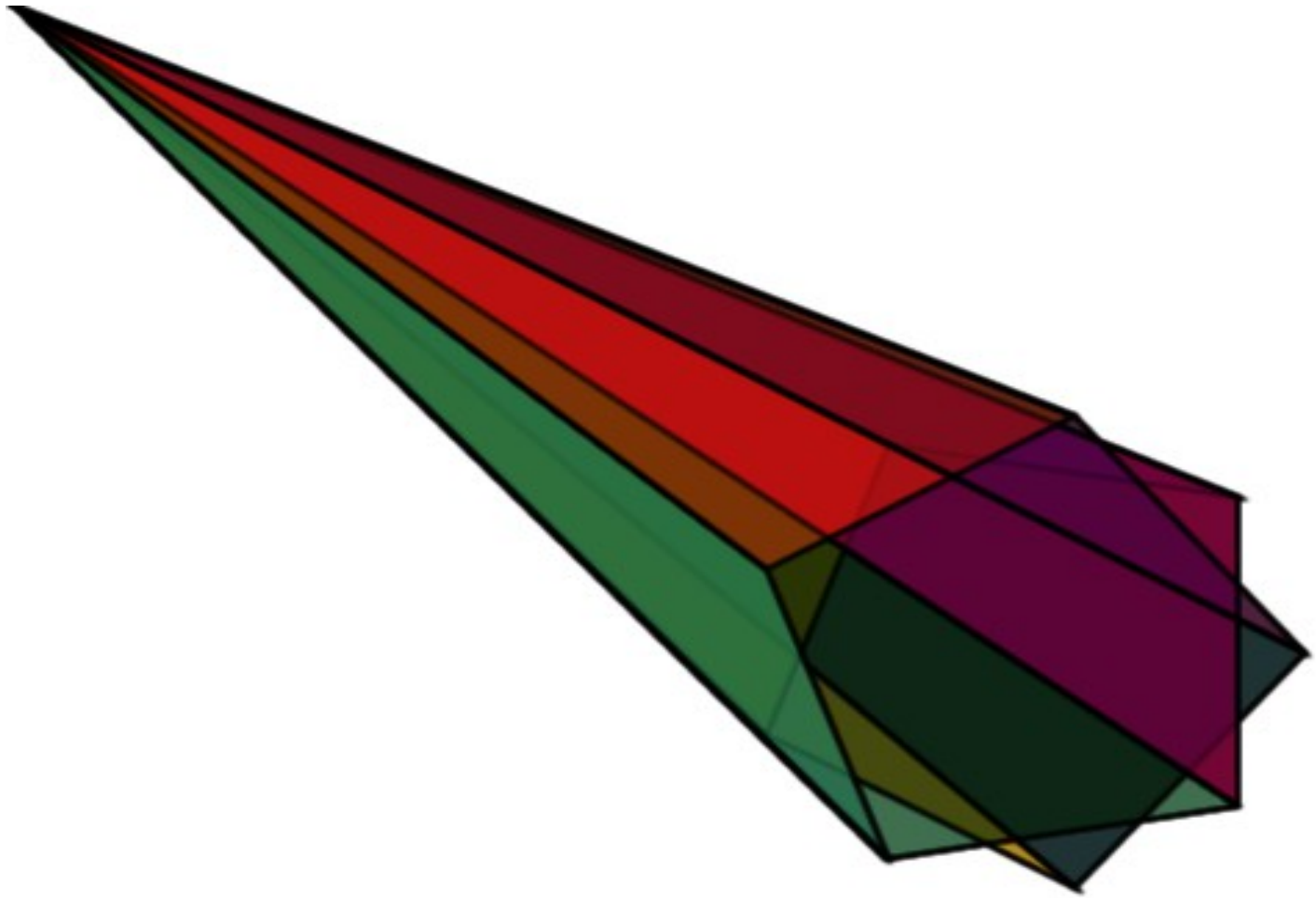


Illustration 43

```
int numPoints = int(map(140, 0, width, 6, 60));
```

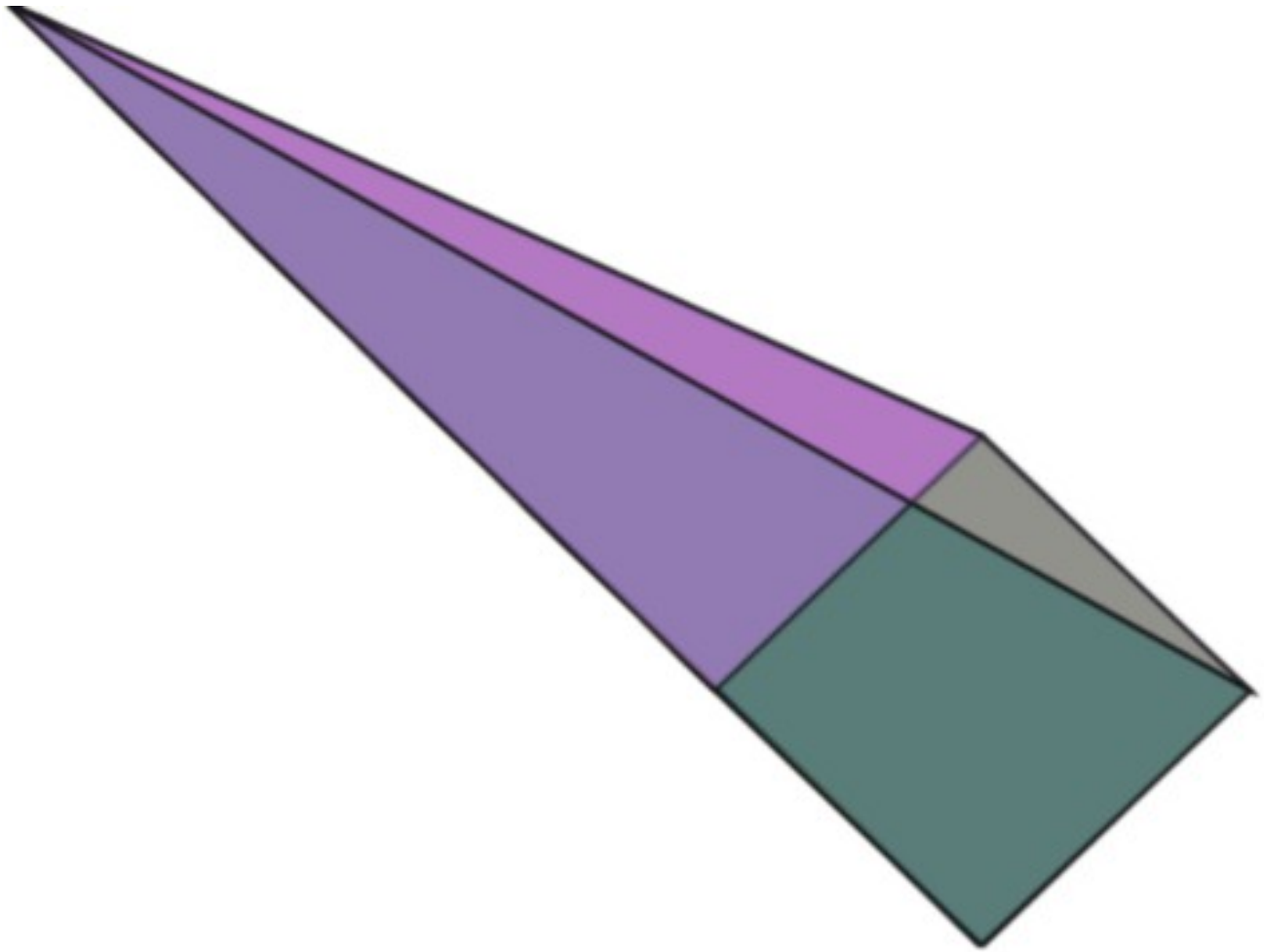


Illustration 44

```
int numPoints = int(map(106, 0, width, 6, 60));
```

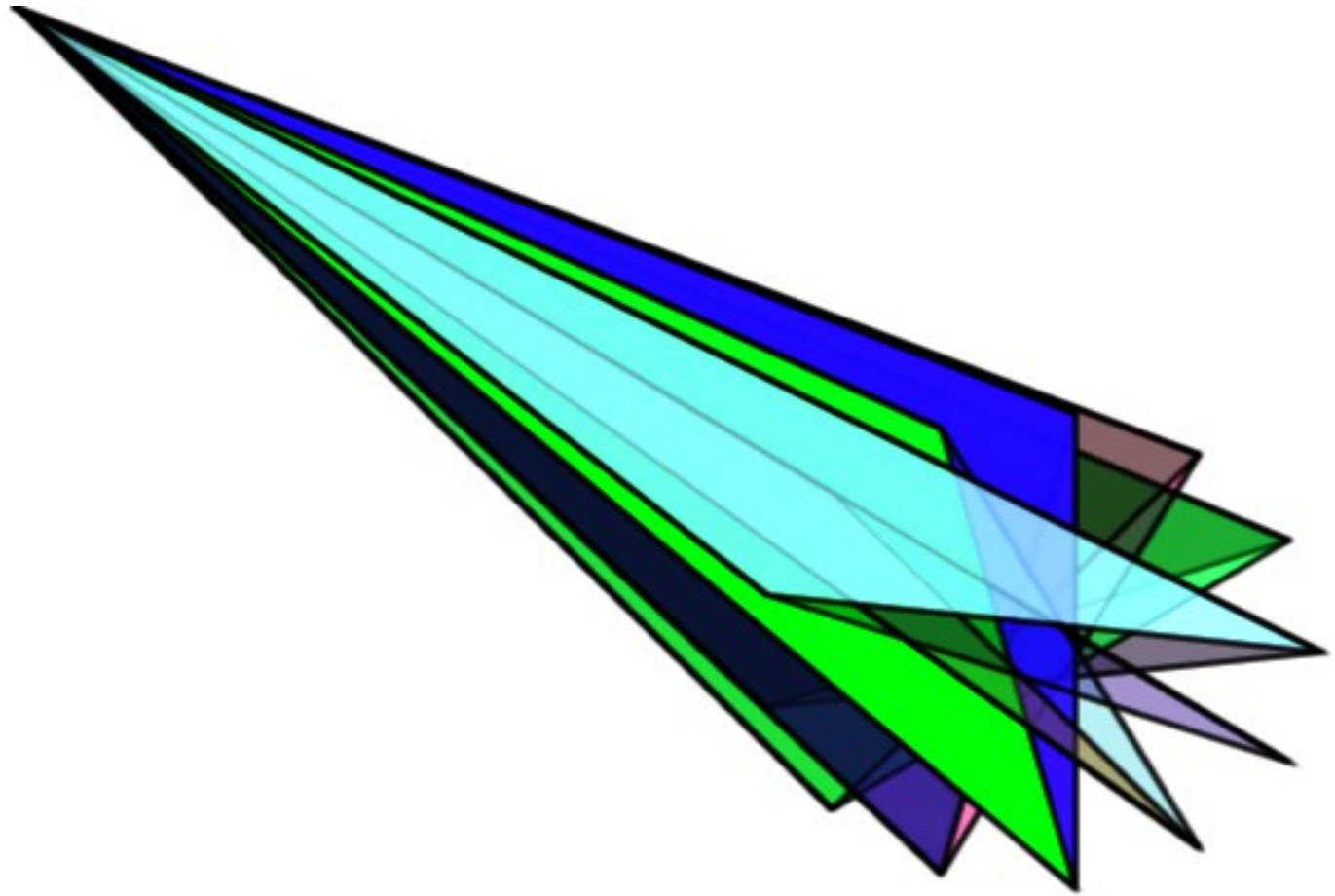


Illustration 45

Shape 18

```
beginShape(PShape.TRIANGLE_STRIP);
 for (int i = 0; i <= numPoints; i++) {
  float px =x + cos(radians(angle)) * insideRadius;
  float py = y + sin(radians(angle)) * insideRadius;
  angle += (angleStep);
  fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
  vertex(px, py);
  px = log(x)+ log(radians(angle)) * outsideRadius;
  py = log(y)+ log(radians(angle)) * outsideRadius;
  angle += (angleStep);
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
 }
 endShape();
```

```
int numPoints = int(map(434, 0, width, 6, 60));
```

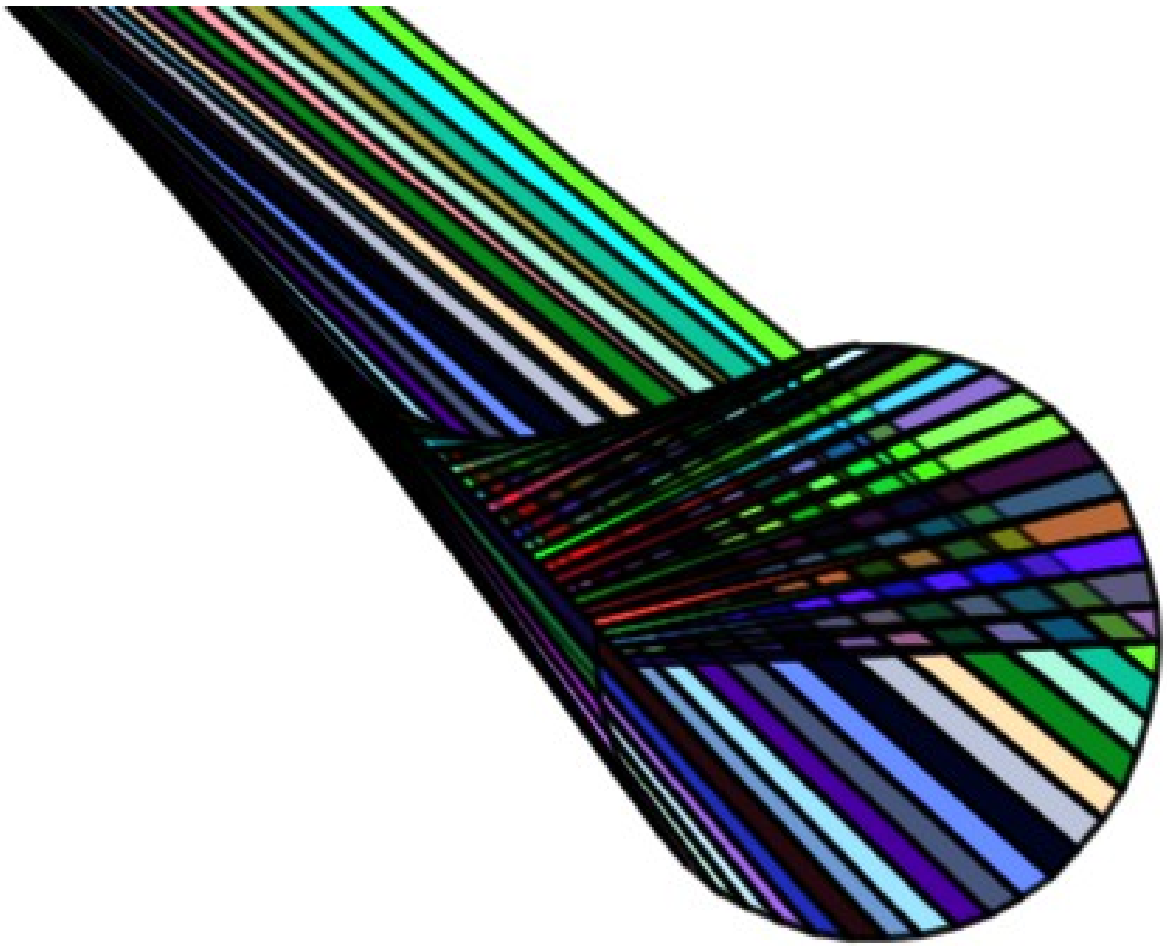


Illustration 46

```
int numPoints = int(map(650, 0, width, 6, 60));
px = log(x)+ log(radians(2*angle)) * outsideRadius;
py = log(y)+ log(radians(2*angle)) * outsideRadius;
```

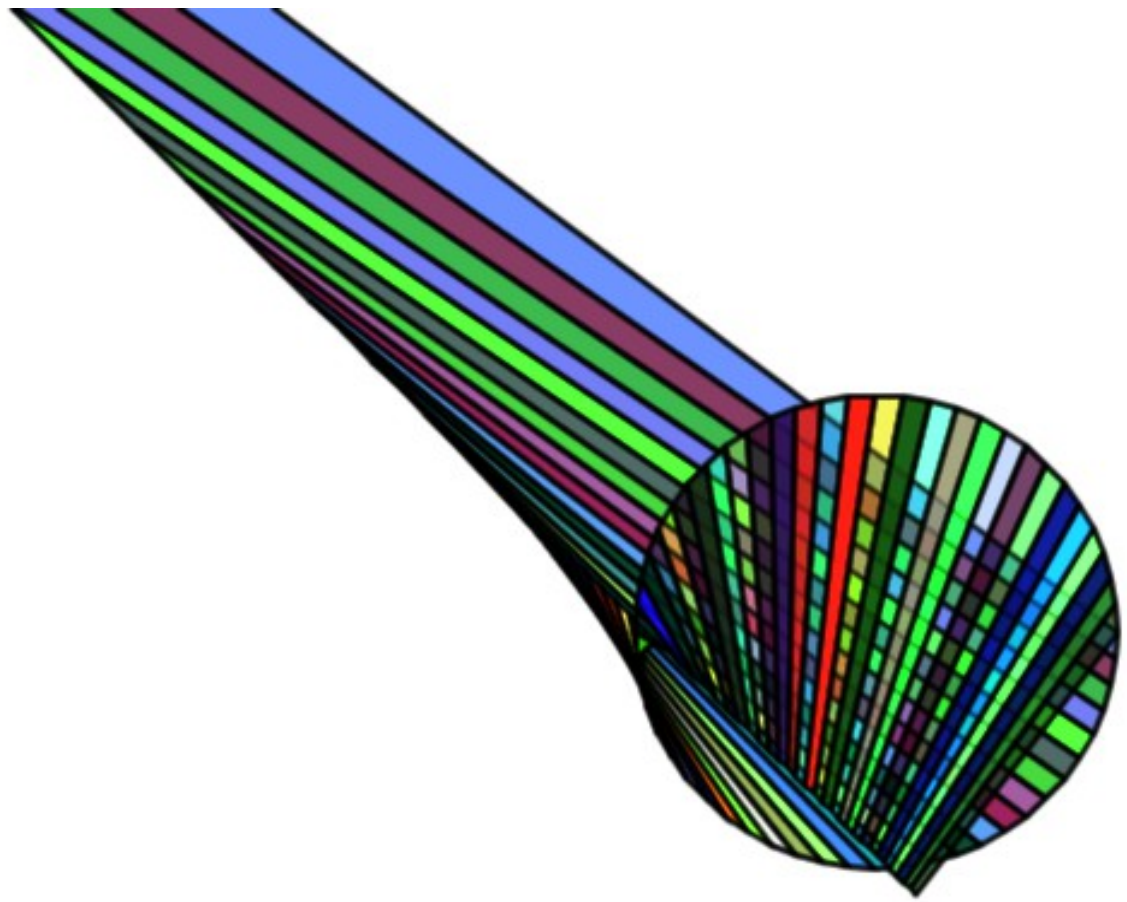


Illustration 47

```
int numPoints = int(map(690, 0, width, 6, 60));
px = log(x)+ log(radians(3*angle)) * outsideRadius;
py = log(y)+ log(radians(3*angle)) * outsideRadius;
```

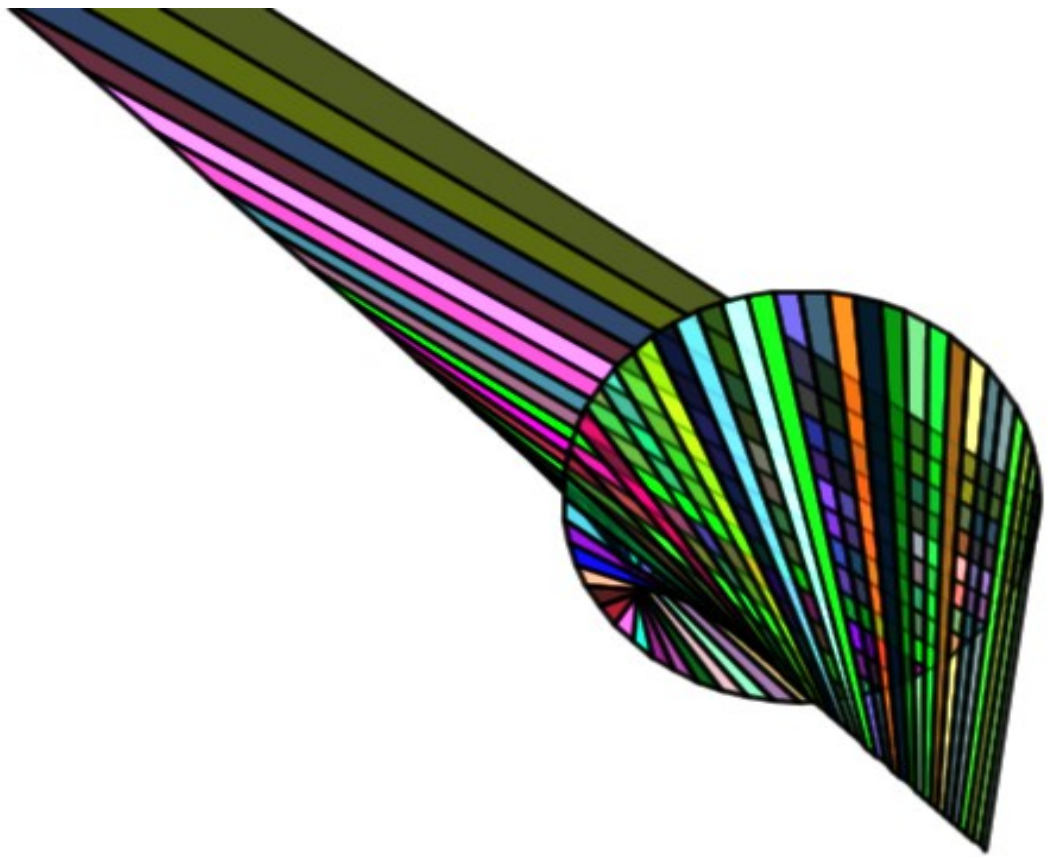


Illustration 48

Shape 19

```
beginShape(PShape.QUAD_STRIP);
for (int i = 0; i <= numPoints; i++) {
  float px =x + cos(radians(angle)) * insideRadius;
  float py = y + sin(radians(angle)) * insideRadius;
  angle += (angleStep);
  fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
  vertex(px, py);
  px = log(x)+ log(radians(1.87*angle)) * outsideRadius; ;
  py = log(y)+ log(radians(1.87*angle)) * outsideRadius;;
  angle += (angleStep);
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
  px = log(x)+ log(radians(1.87*angle)) * outsideRadius; ;
  py = log(y)+ log(radians(1.87*angle)) * outsideRadius;;
  angle += (angleStep);
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
  px = log(x)+ log(radians(1.87*angle)) * outsideRadius; ;
  py = log(y)+ log(radians(1.87*angle)) * outsideRadius;;
  angle += (angleStep);
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
}
endShape();
```

```
int numPoints = int(map(695, 0, width, 6, 60));
```

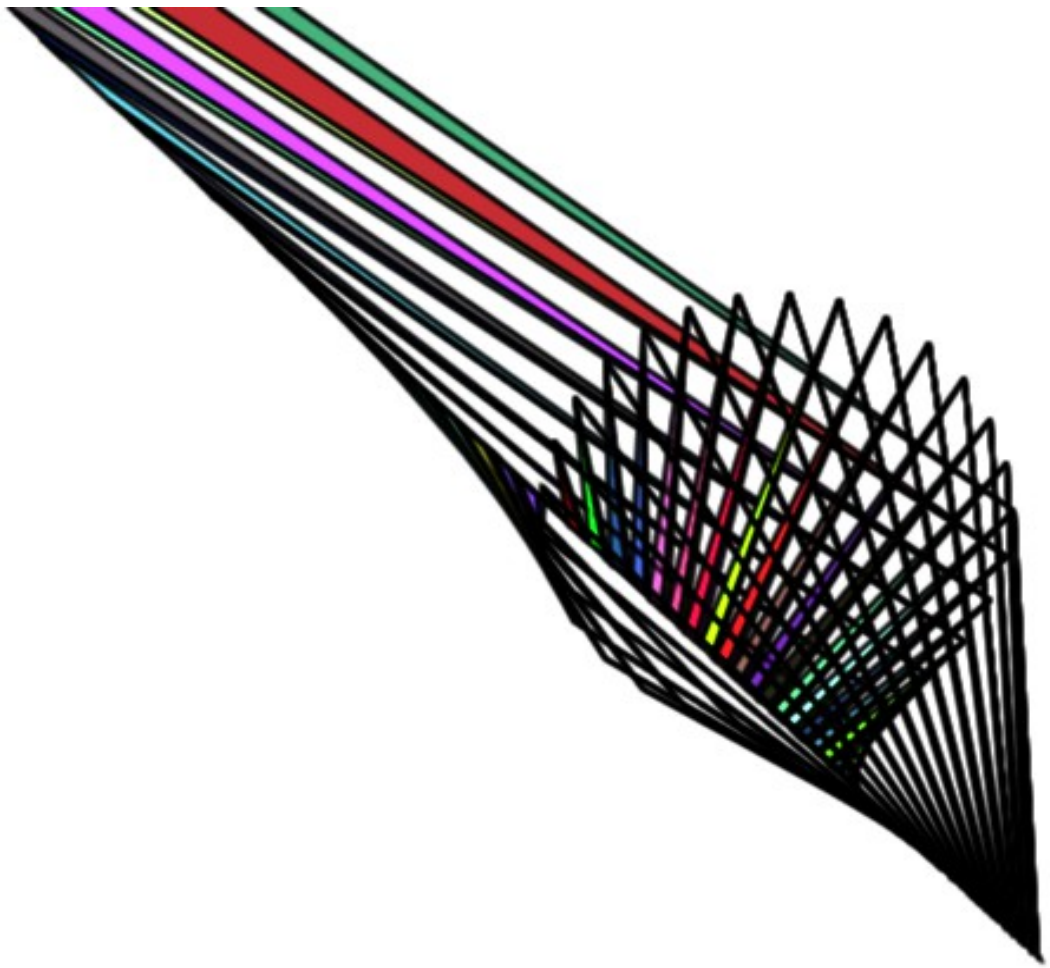


Illustration 49

```
int numPoints = int(map(13, 0, width, 6, 60));
```

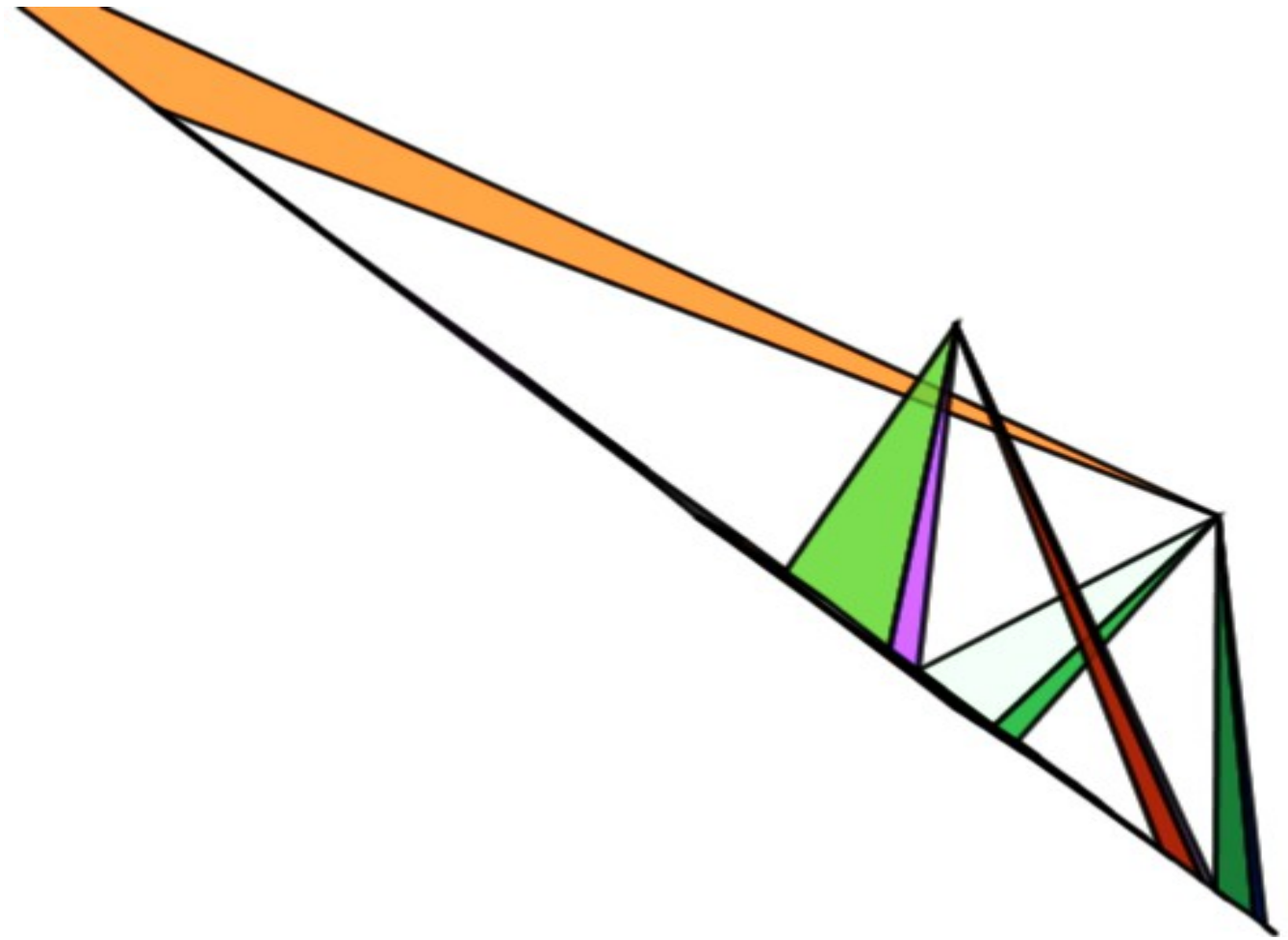


Illustration 50

```
int numPoints = int(map(13, 0, width, 6, 60));
beginShape(PShape.TRIANGLE_STRIP);
```

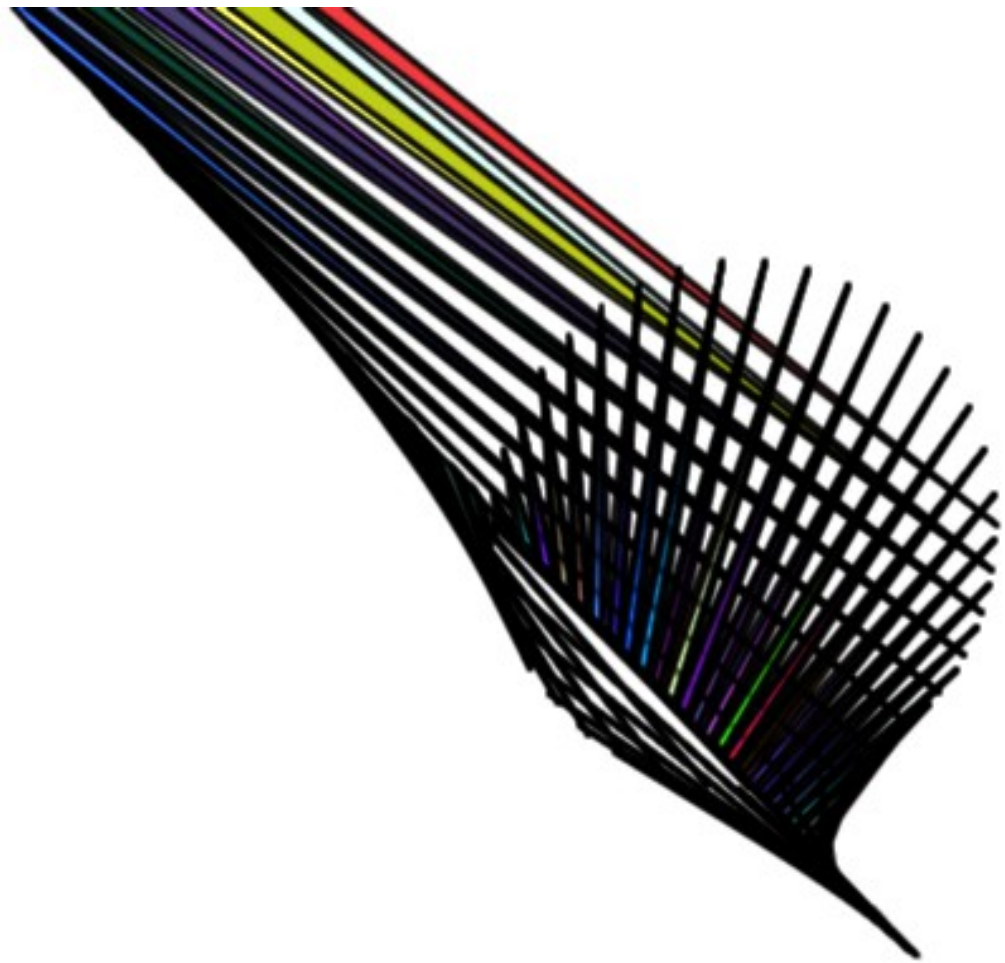


Illustration 51

```
beginShape(PShape.TRIANGLE_STRIP);                    Shape 20
 for (int i = 0; i <= numPoints; i++) {
   float px =x + cos(radians(angle)) * insideRadius;
   float py = y + sin(radians(angle)) * insideRadius;
   angle += (angleStep);
   fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
   vertex(px, py);
   px = sin(x)+ cos(radians(2*angle)) * outsideRadius; ;
   py = sin(y)+ cos(radians(2*angle)) * outsideRadius;;
   angle += (angleStep);
   fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
   vertex(px, py);
 }
 endShape();
```

```
int numPoints = int(map(633, 0, width, 6, 60));
```

Illustration 52

```
beginShape(PShape.TRIANGLE_STRIP);                    Shape 21
 for (int i = 0; i <= numPoints; i++) {
   float px =x + sin(radians(angle)) * insideRadius;
   float py = y + cos(radians(angle)) * insideRadius;
   angle += (angleStep);
   fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
   vertex(px, py);
   px = x+ sin(radians(angle)) * outsideRadius; ;
   py = y+ cos(radians(angle)) * outsideRadius;;
   angle += (angleStep);
   fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
   vertex(px, py);
 }
 endShape();
```

```
int numPoints = int(map(45, 0, width, 6, 60));
```

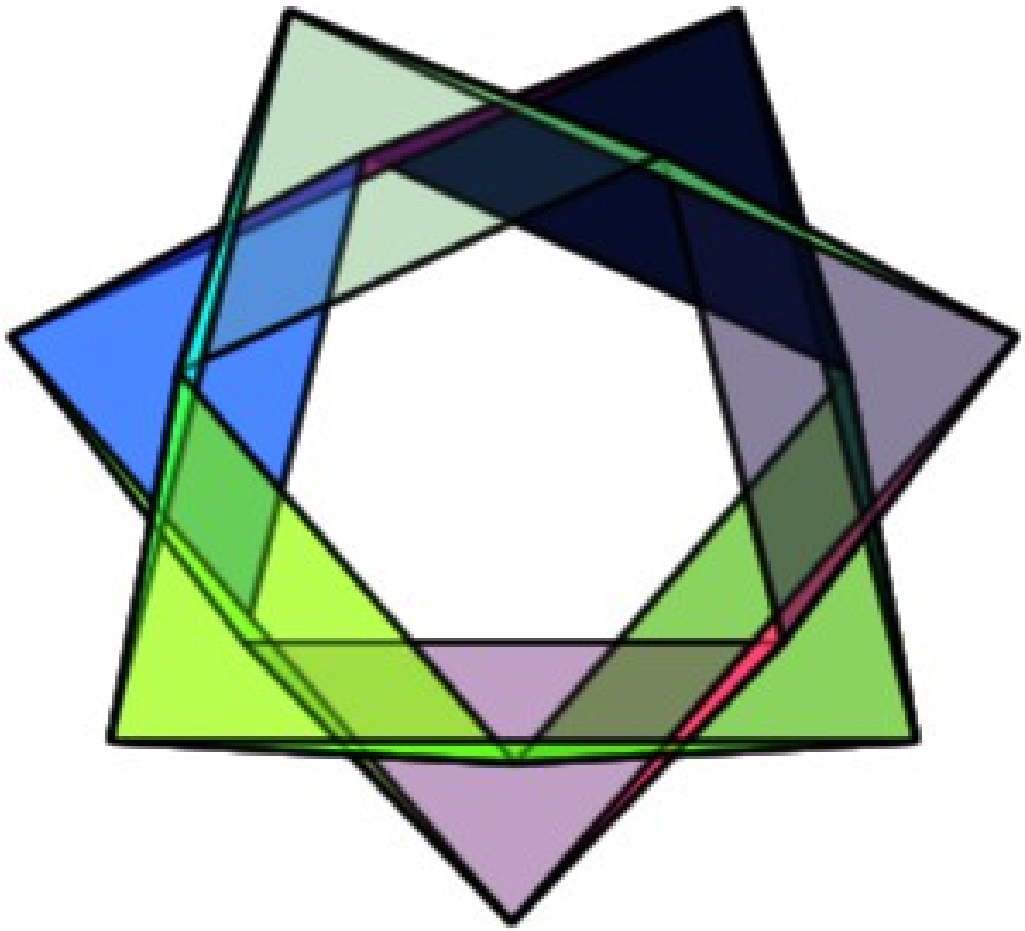


Illustration 53

```
int numPoints = int(map(115, 0, width, 6, 60));
```

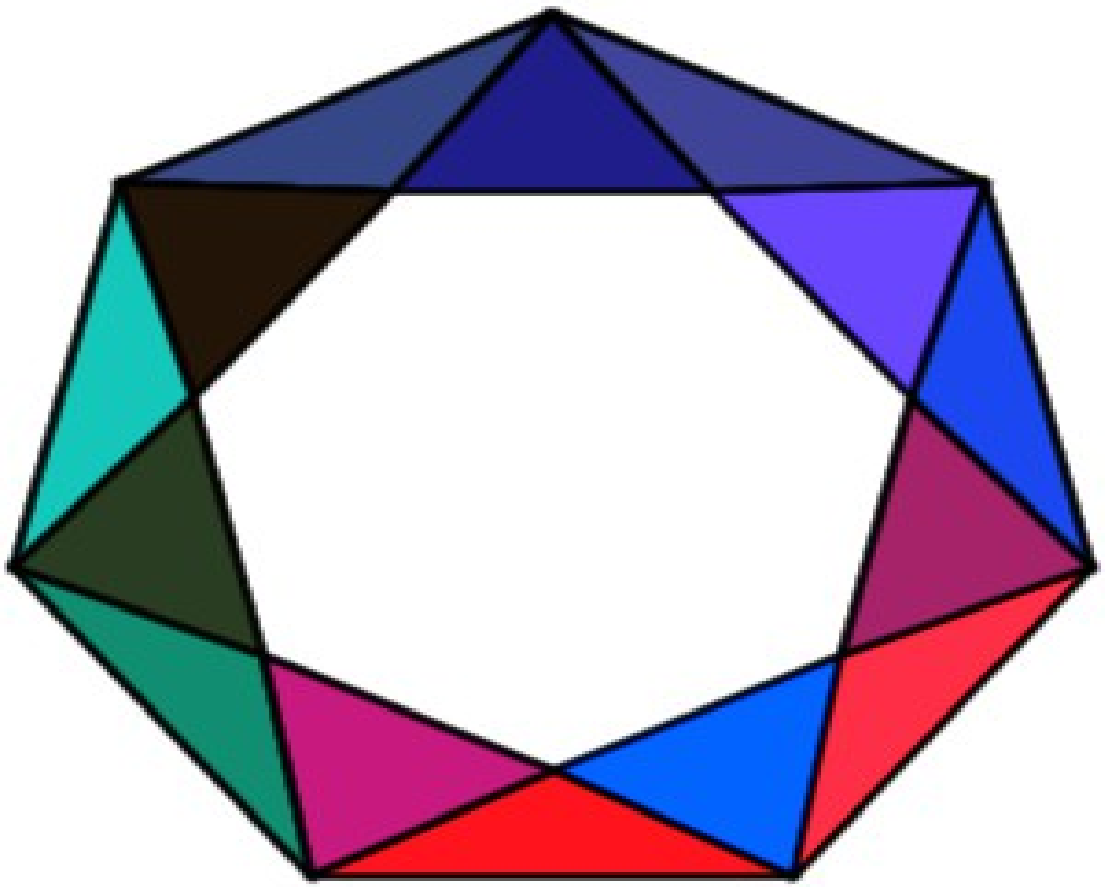


Illustration 54

Shape 22

```
beginShape(PShape.QUAD_STRIP);
 for (int i = 0; i <= numPoints; i++) {
  float px =x + cos(radians(angle)) * insideRadius;
  float py = y + sin(radians(angle)) * outsideRadius;
  angle += (angleStep);
  fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
  vertex(px, py);
  px = x+cos(radians(angle)) * outsideRadius;
  py = y+sin(radians(angle)) * outsideRadius;
  angle += (angleStep);
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
  px = x+cos(radians(angle)) * outsideRadius;
 py = y+sin(radians(angle)) * outsideRadius;
 angle += (angleStep);
 fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
 vertex(px, py);
  px = x+cos(radians(angle)) * insideRadius;
 py = y+sin(radians(angle)) * insideRadius;
  angle += (angleStep);
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
 vertex(px, py);
}
endShape();
```

```
int numPoints = int(map(30, 0, width, 6, 60));
```

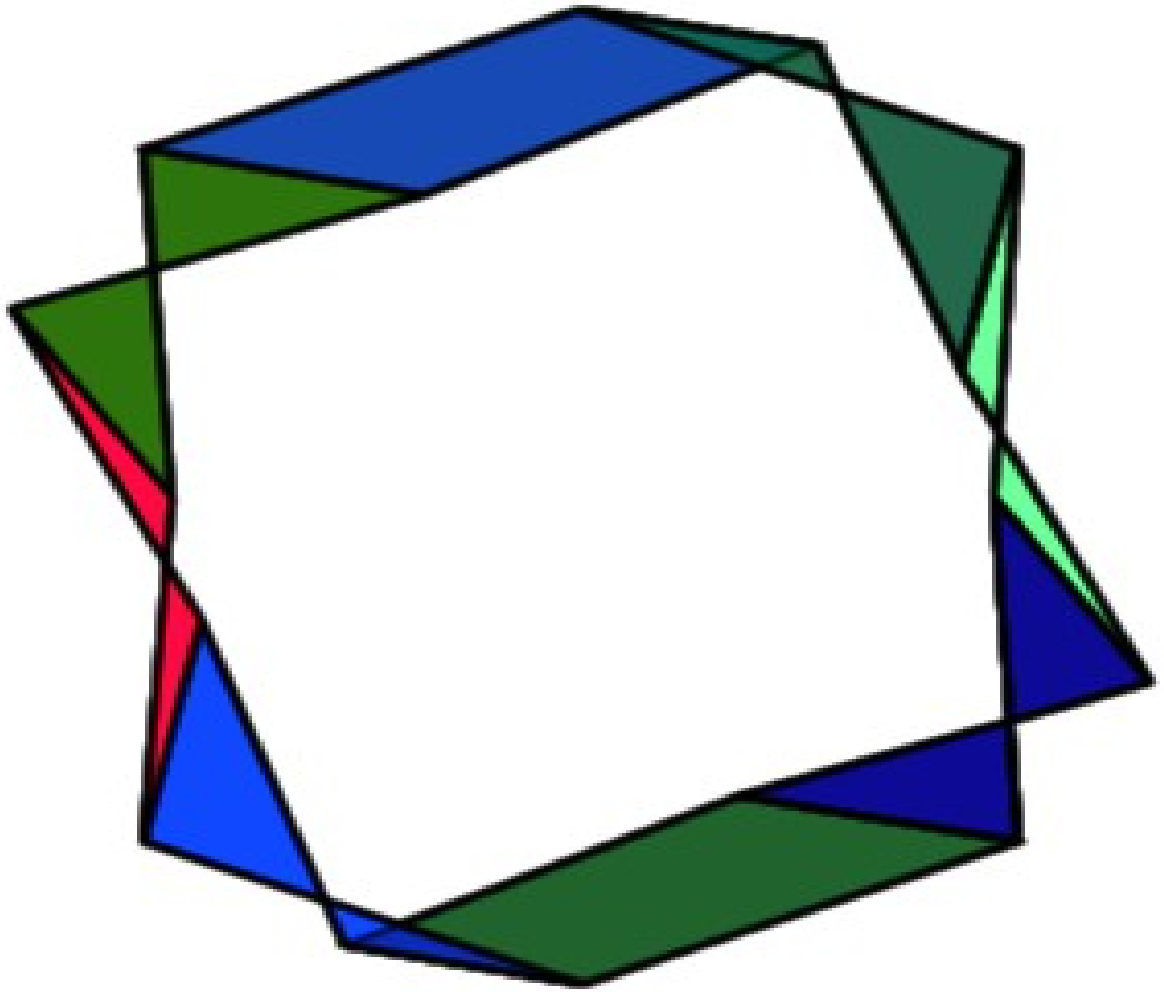


Illustration 55

```
int numPoints = int(map(660, 0, width, 6, 60));
```

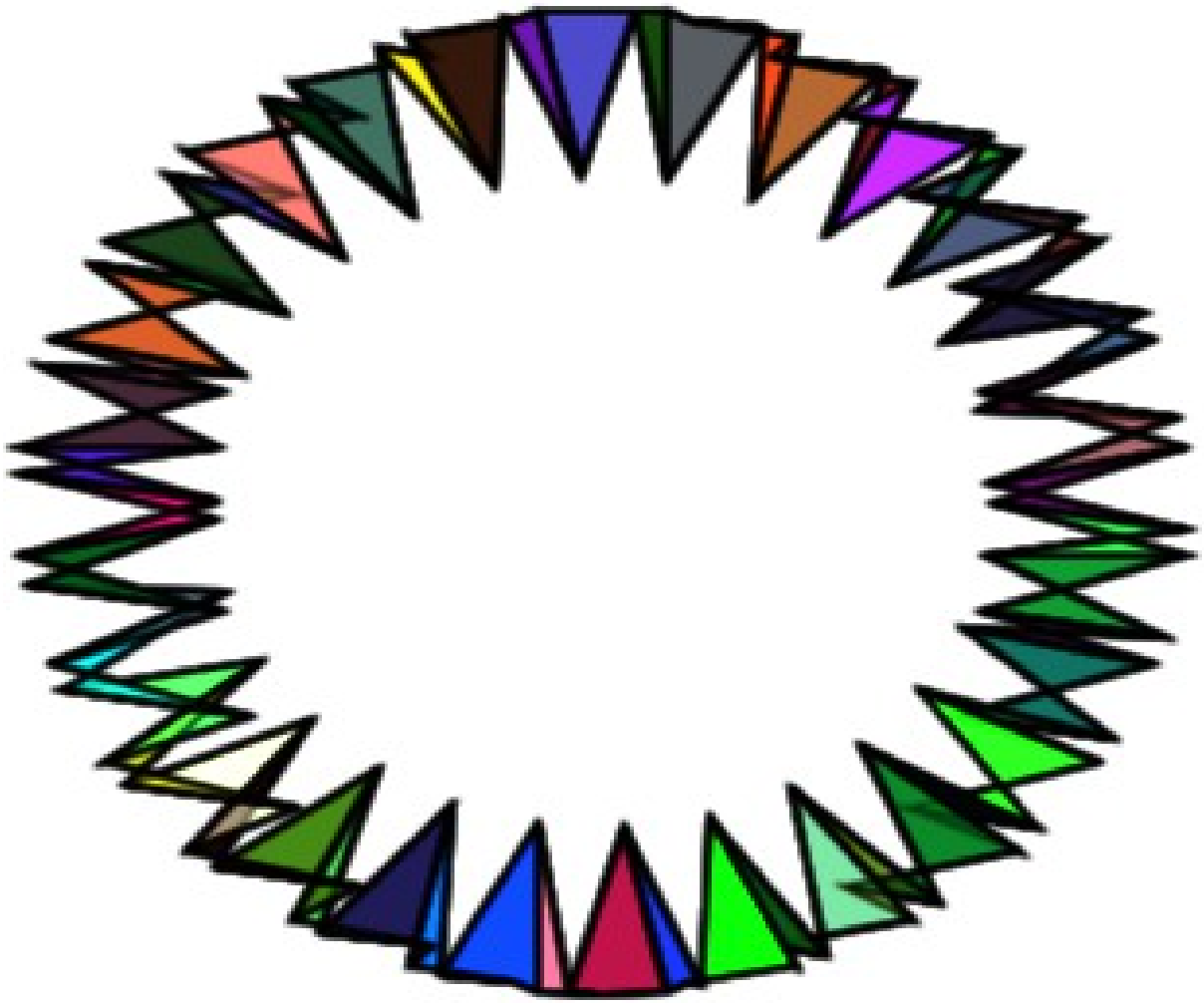


Illustration 56

```
int numPoints = int(map(702, 0, width, 6, 60));
```

Illustration 57

Shape 23

```
beginShape(PShape.TRIANGLE_STRIP);
  for (int i = 0; i <= numPoints; i++) {
    float px =x + cos(radians(angle)) * insideRadius;
    float py = y + sin(radians(angle)) * outsideRadius;
    angle += (angleStep);
    fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
    vertex(px, py);
    px = x+cos(radians(angle)) * outsideRadius;
    py = y+sin(radians(angle)) * insideRadius;
    angle += (angleStep);
    fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
    vertex(px, py);
    px = x+cos(radians(angle)) * outsideRadius;
   py = y+sin(radians(angle)) * insideRadius;
   angle += (angleStep);
   fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
   vertex(px, py);
    px = x+cos(radians(angle)) * outsideRadius;
   py = y+sin(radians(angle)) * insideRadius;
    angle += (angleStep);
    fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
   vertex(px, py);
  } endShape();
```

```
int numPoints = int(map(702, 0, width, 6, 60));
```

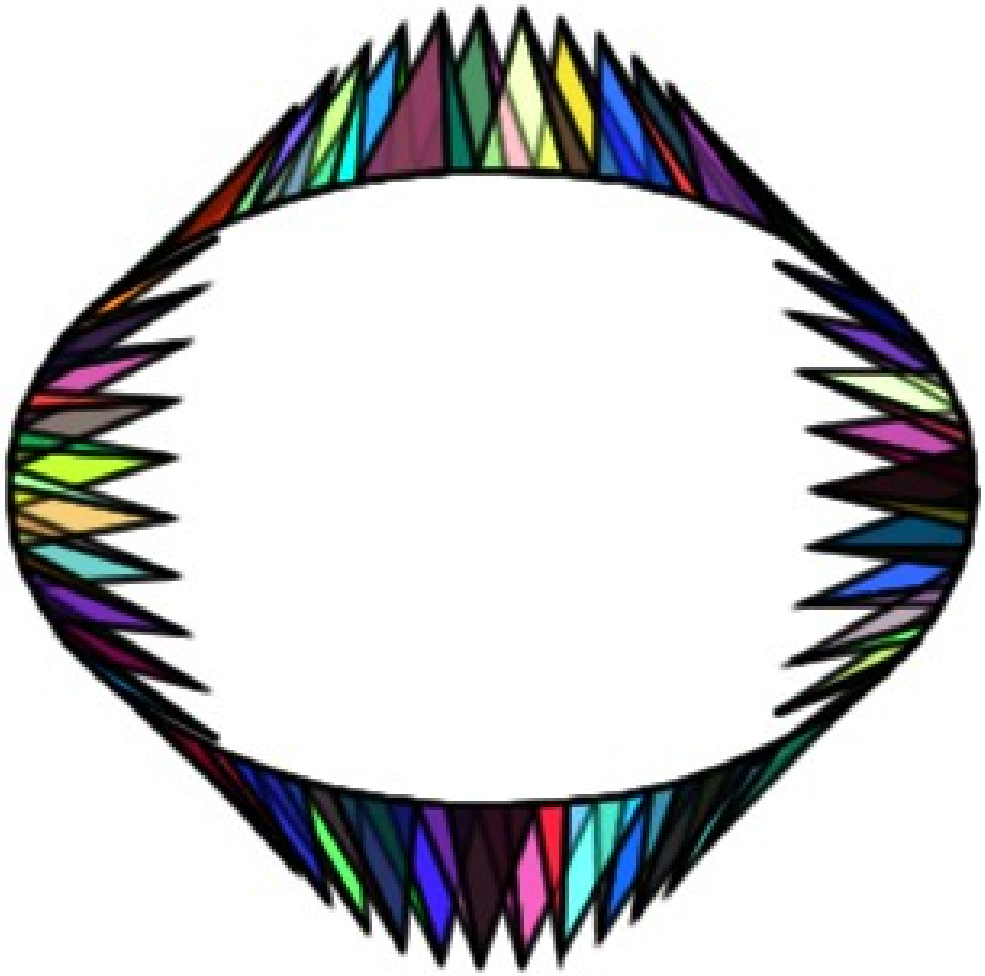


Illustration 58

```
int numPoints = int(map(142, 0, width, 6, 60));
```

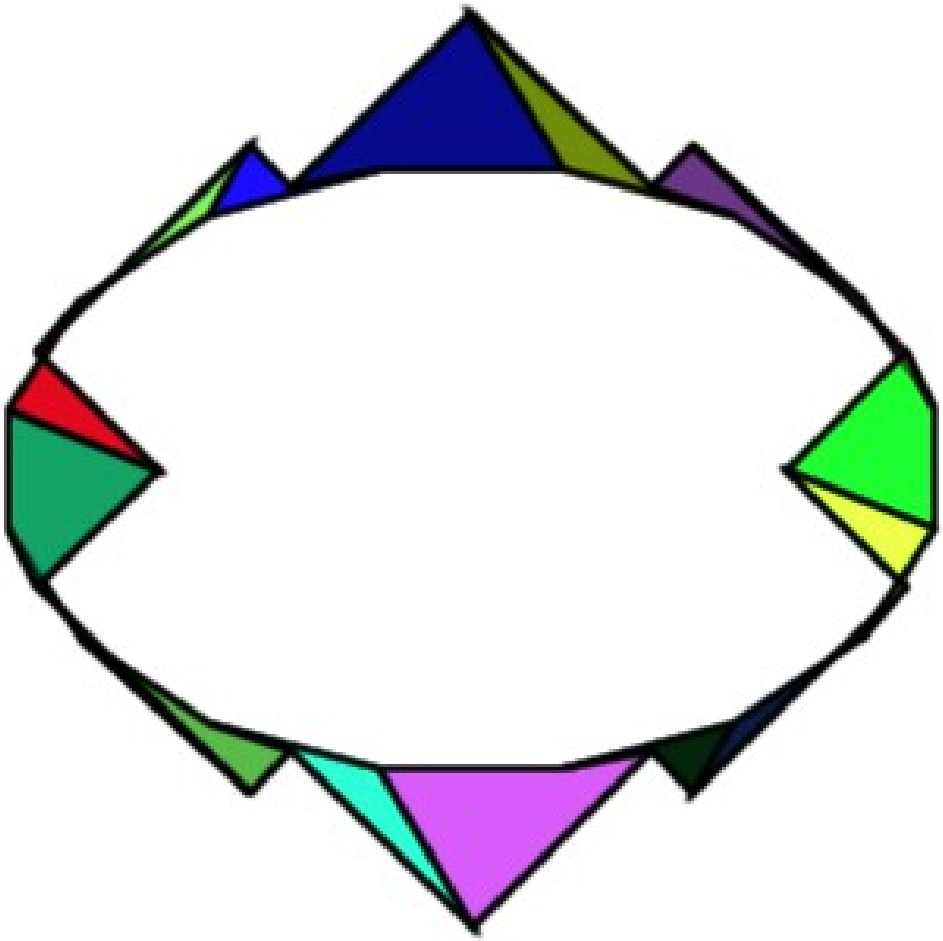


Illustration 59

int numPoints = int(map(312, 0, width, 6, 60));

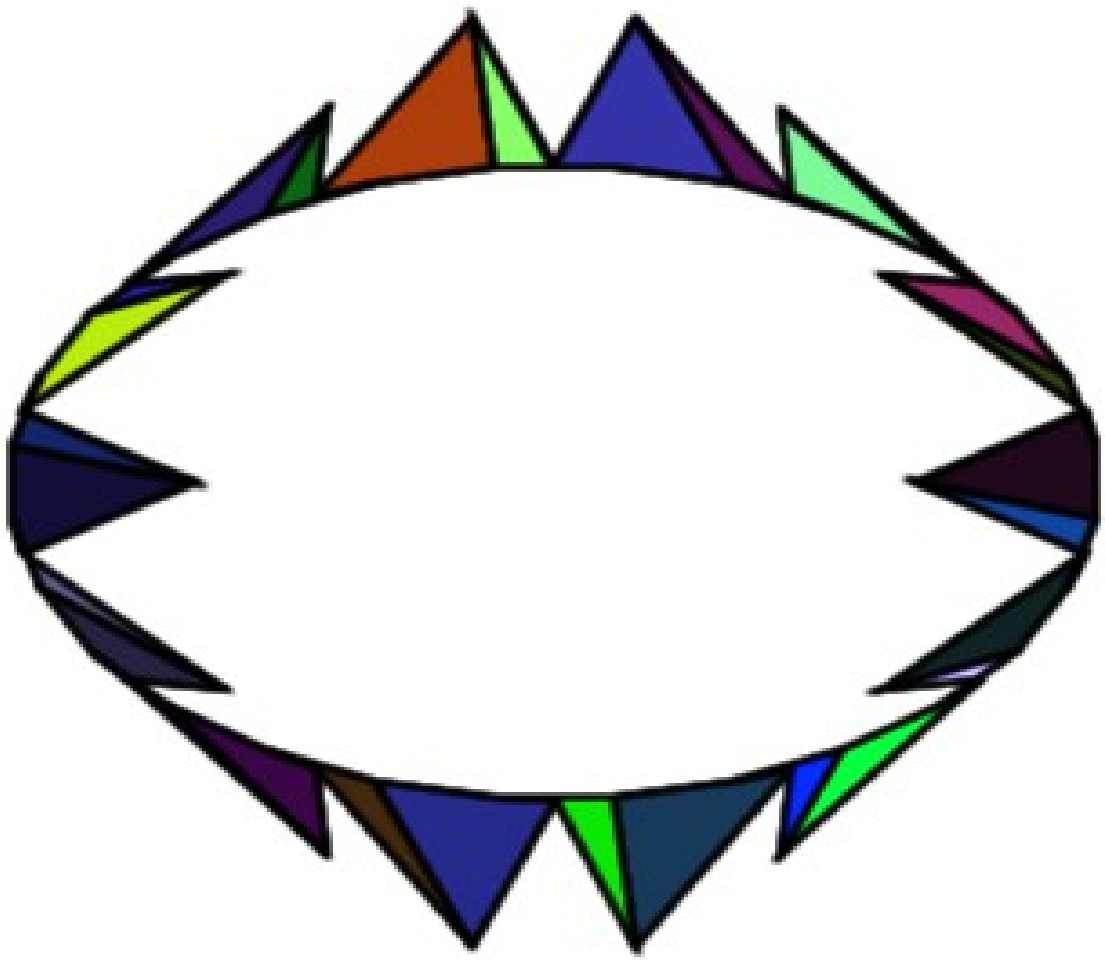


Illustration 60

```
int numPoints = int(map(738, 0, width, 6, 60));
```

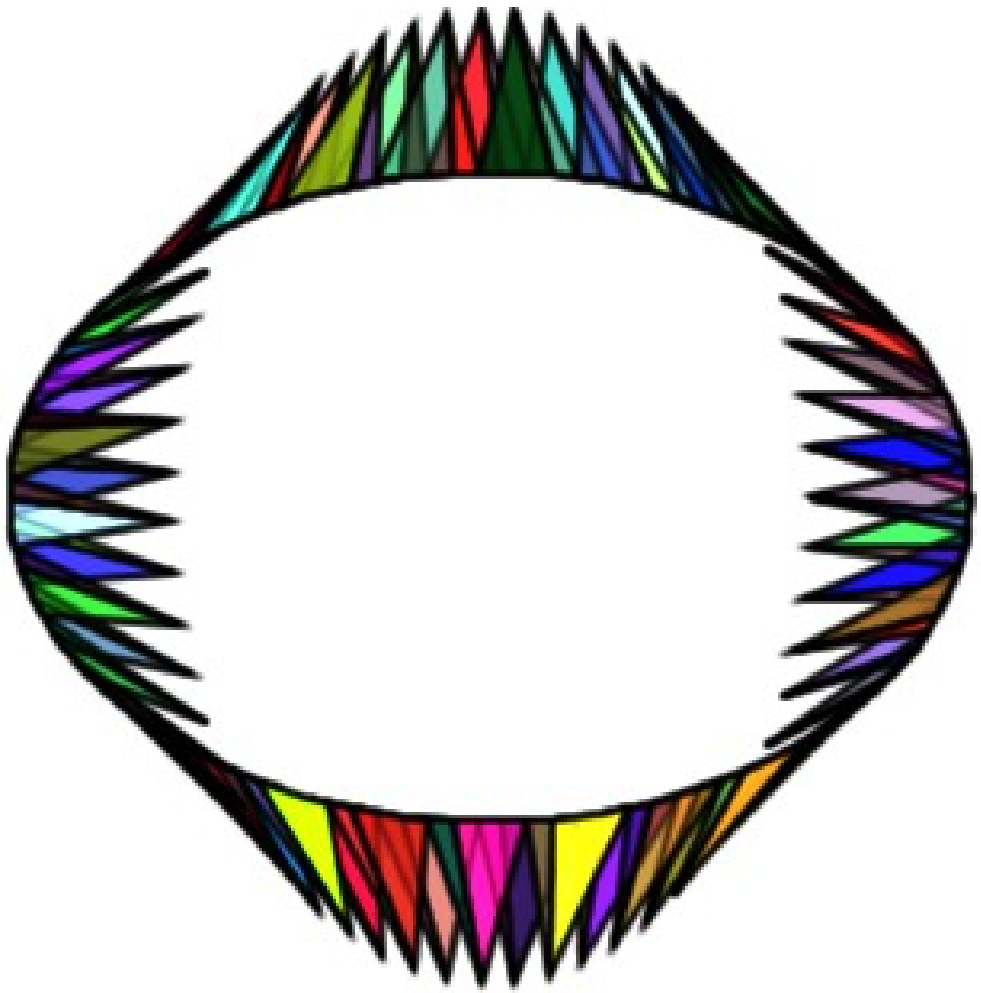


Illustration 61

```
int numPoints = int(map(86, 0, width, 6, 60));
```

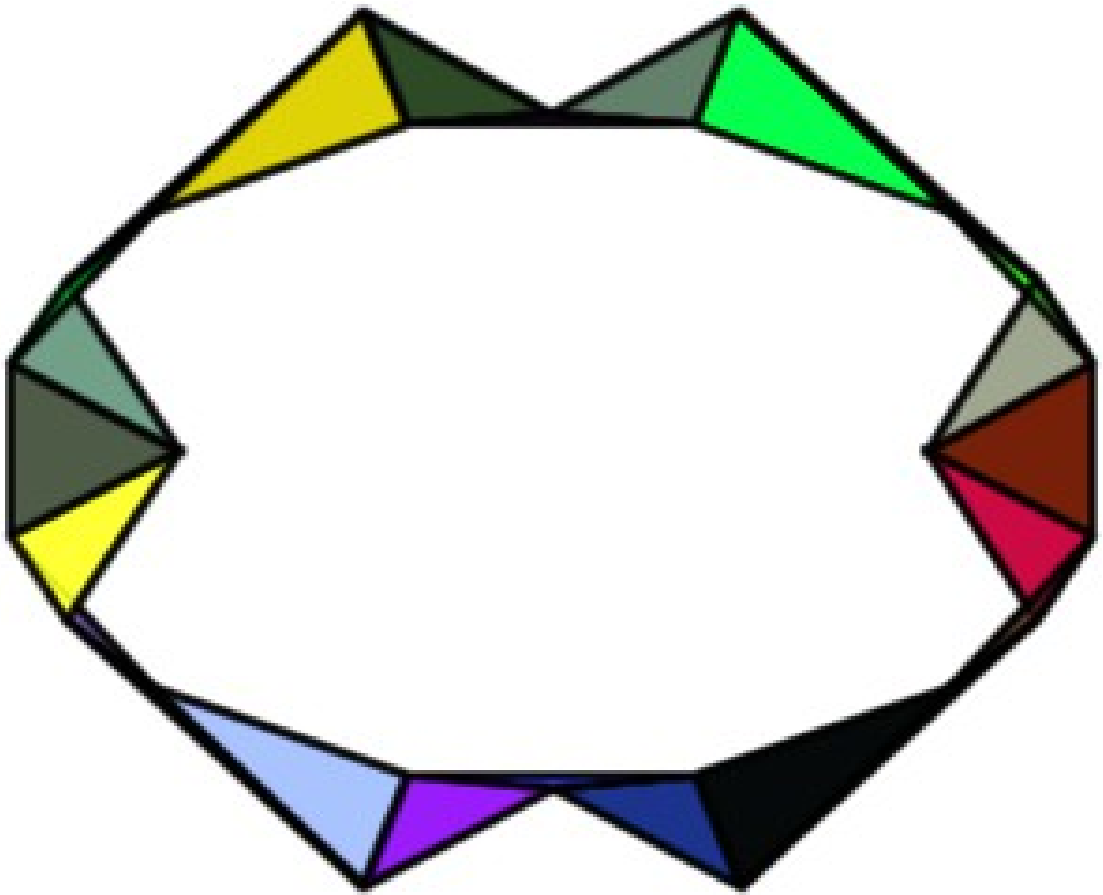


Illustration 62

Shape 24

```
beginShape(PShape.TRIANGLE_STRIP);
 for (int i = 0; i <= numPoints; i++) {
  float px =x + cos(radians(angle)) * outsideRadius;
  float py = y + sin(radians(angle)) * insideRadius;
  angle += (angleStep);
  fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
  vertex(px, py);
  px = x+cos(radians(angle)) * insideRadius;
  py = y+sin(radians(angle)) * outsideRadius;
  angle += (angleStep);
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
  px = x+cos(radians(angle)) * insideRadius;
 py = y+sin(radians(angle)) * outsideRadius;
 angle += (angleStep);
 fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
 vertex(px, py);
  px = x+cos(radians(angle)) * outsideRadius;
 py = y+sin(radians(angle)) * insideRadius;
  angle += (angleStep);
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
 vertex(px, py);
}endShape();
```

```
int numPoints = int(map(376, 0, width, 6, 60));
```

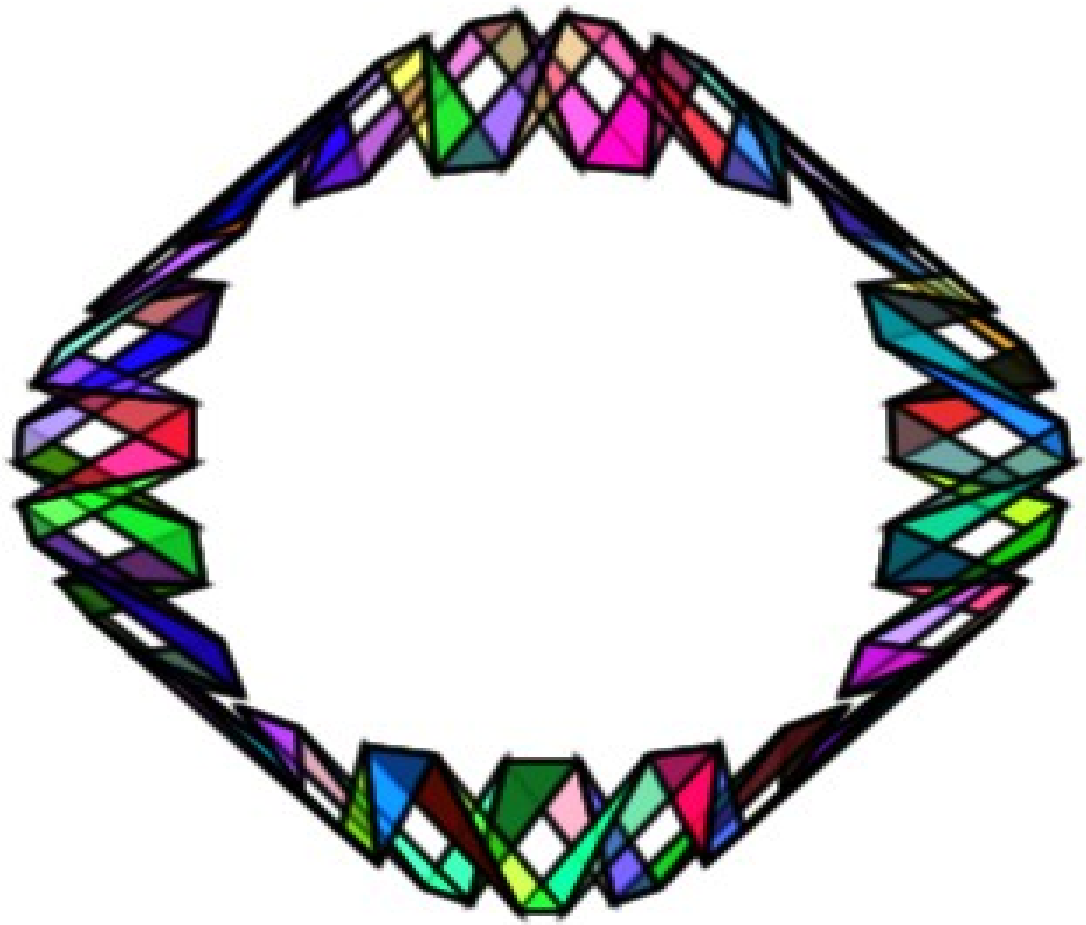


Illustration 63

```
int numPoints = int(map(376, 0, width, 6, 60));
```

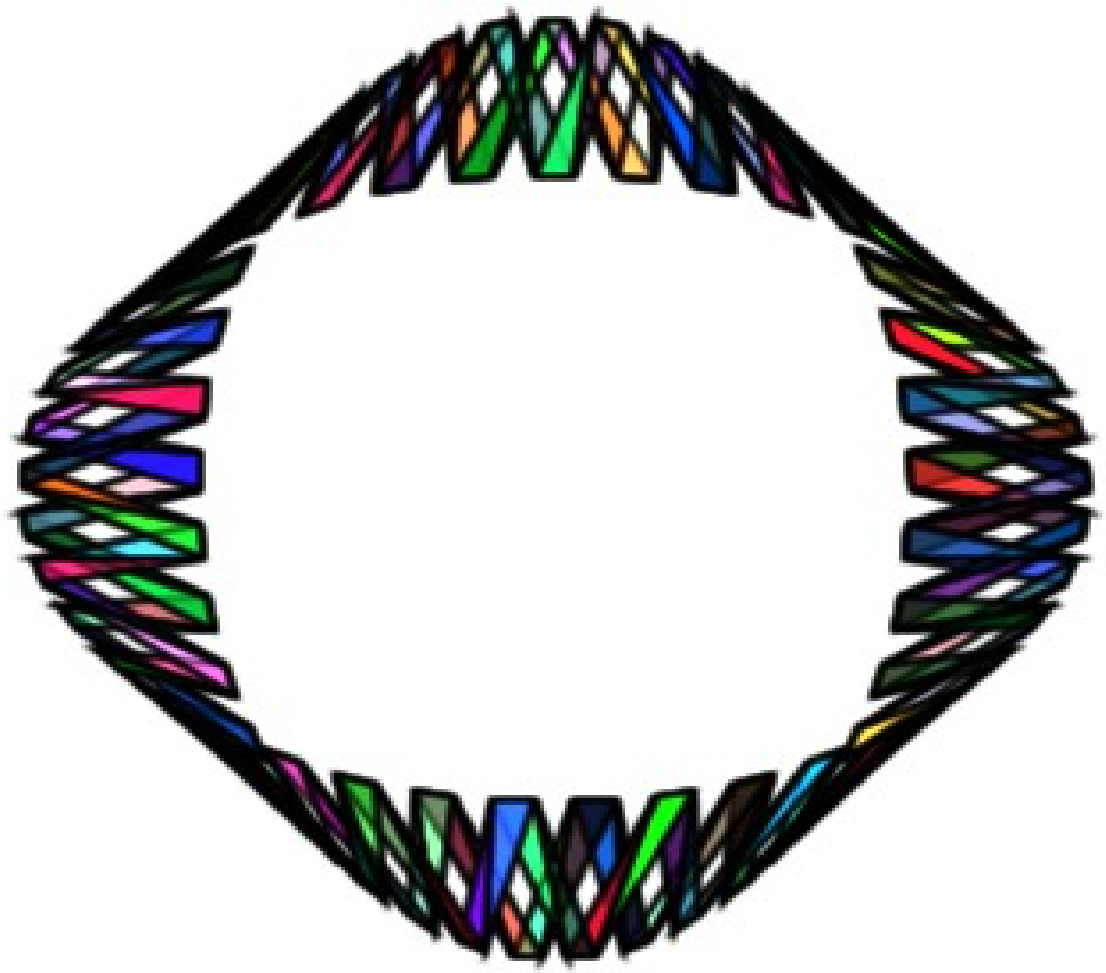


Illustration 64

```
int numPoints = int(map(135, 0, width, 6, 60));
```

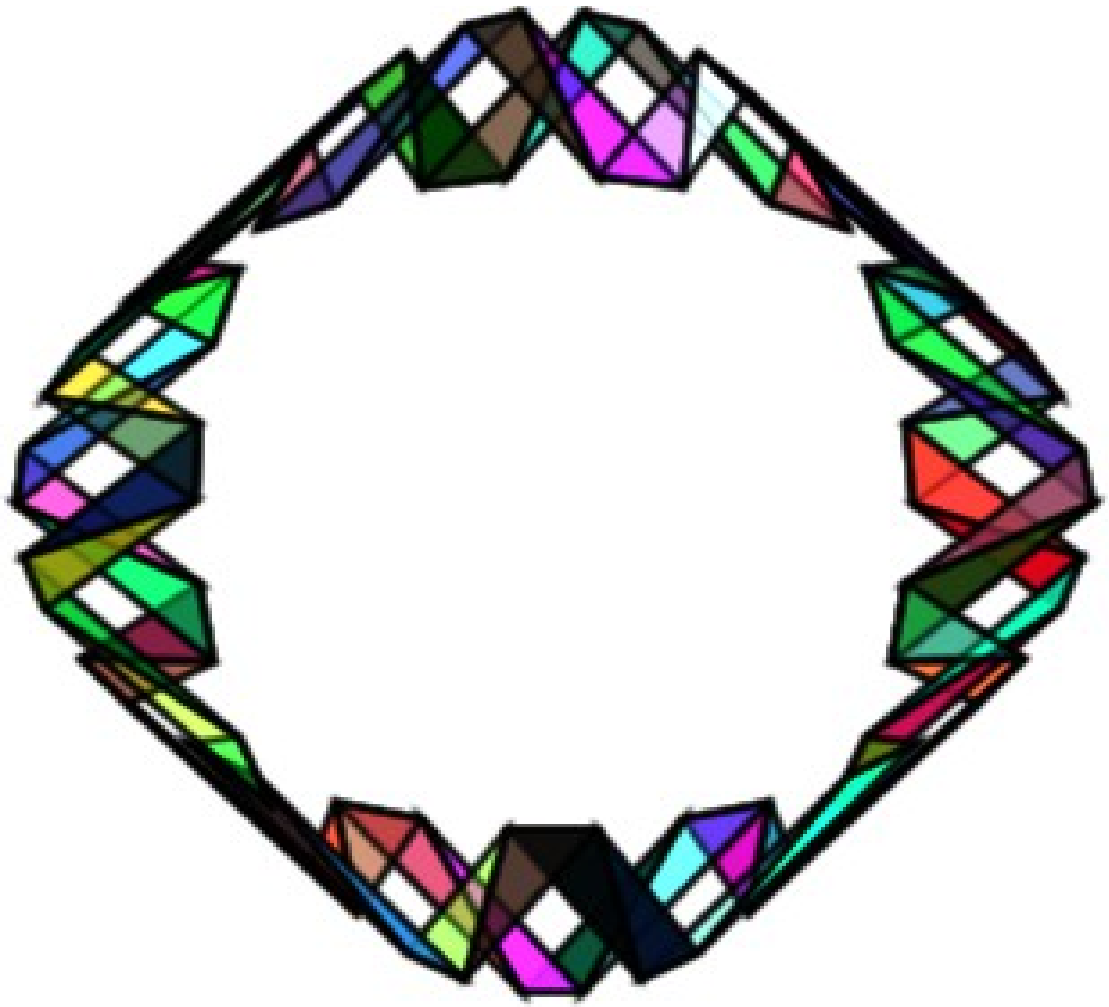


Illustration 65

```
int numPoints = int(map(135, 0, width, 6, 60));
```

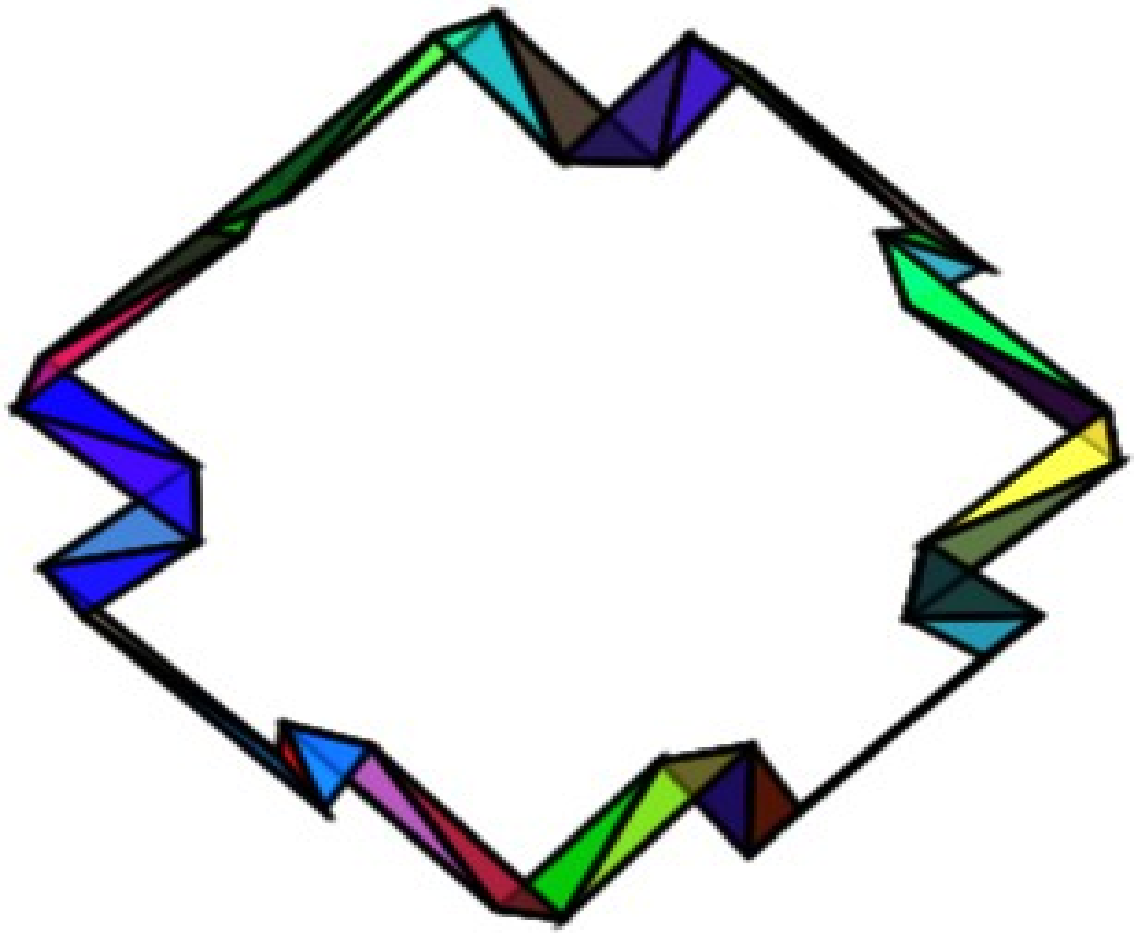


Illustration 66

```
float insideRadius = 10;
int numPoints = int(map(135, 0, width, 6, 60));
```

Illustration 67

```
float insideRadius = 10;
int numPoints = int(map(42, 0, width, 6, 60));
```

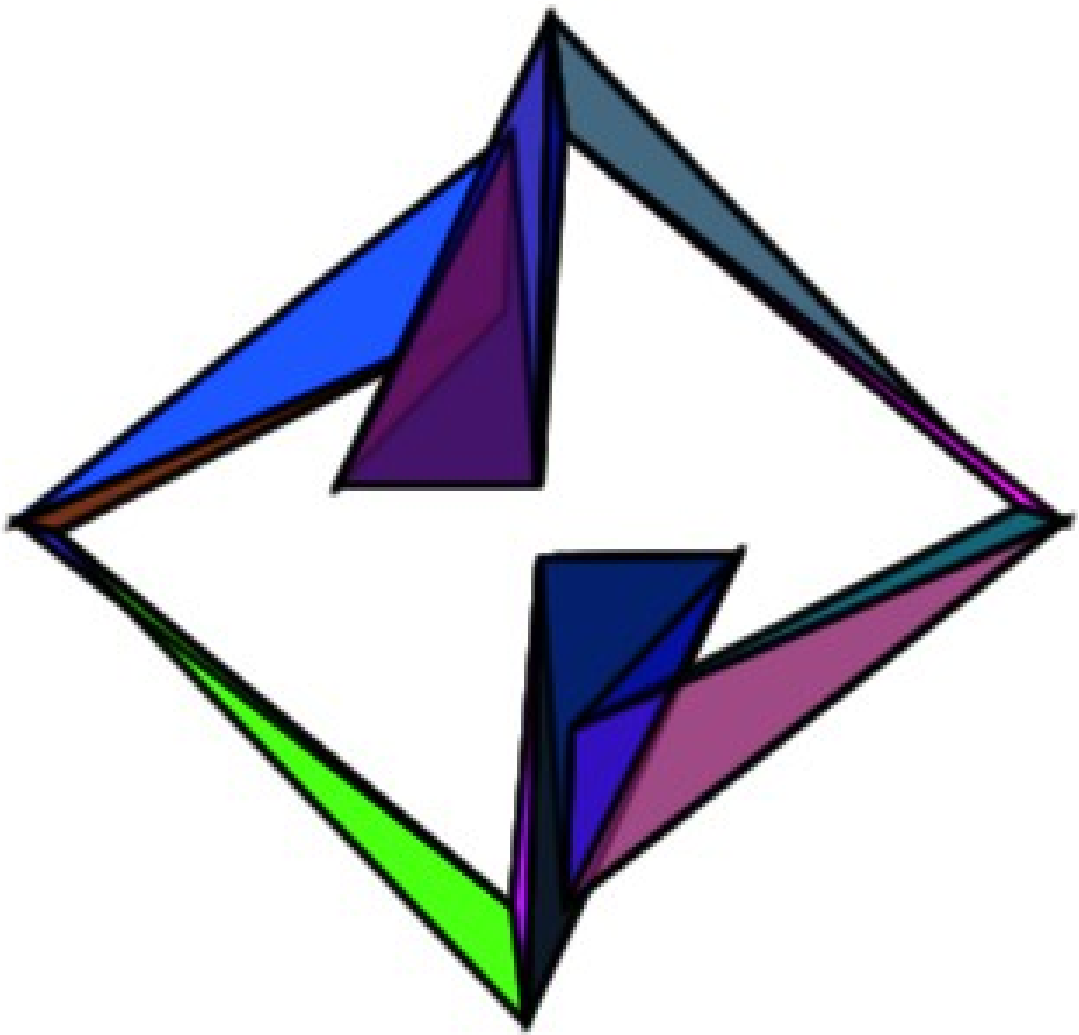


Illustration 68

```
float insideRadius = 10;
int numPoints = int(map(42, 0, width, 6, 60));
```

Illustration 69

```
float insideRadius = 10;
int numPoints = int(map(72, 0, width, 6, 60));
```

Illustration 70

```
float insideRadius = 10;
int numPoints = int(map(171, 0, width, 6, 60));
```

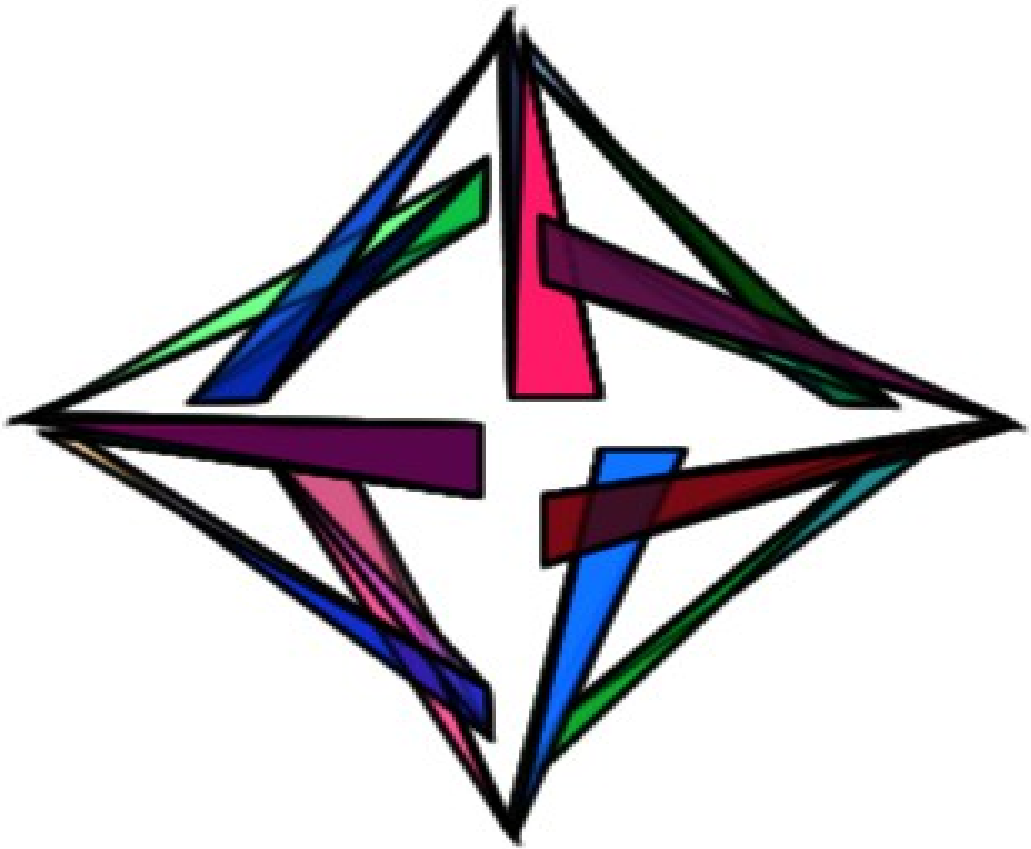


Illustration 71

Shape 25

```
beginShape(PShape.TRIANGLE_STRIP);
 for (int i = 0; i <= numPoints; i++) {
   float px =x + cos(radians(angle)) * insideRadius;
   float py = y + sin(radians(angle)) * insideRadius;
   angle += (angleStep);
   fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
   vertex(px, py);
   px = x+cos(radians(angle)) * insideRadius;
   py = y+sin(radians(angle)) * outsideRadius;
   angle += (angleStep);
   fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
   vertex(px, py);
   px = x+cos(radians(angle)) * outsideRadius;
   py = y+sin(radians(angle)) * insideRadius;
   angle += (angleStep);
   fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
   vertex(px, py);
   px = x+cos(radians(angle)) * outsideRadius;
   py = y+sin(radians(angle)) * outsideRadius;
   angle += (angleStep);
   fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
   vertex(px, py);
 }
endShape();
```

```
int numPoints = int(map(222, 0, width, 6, 60));
```

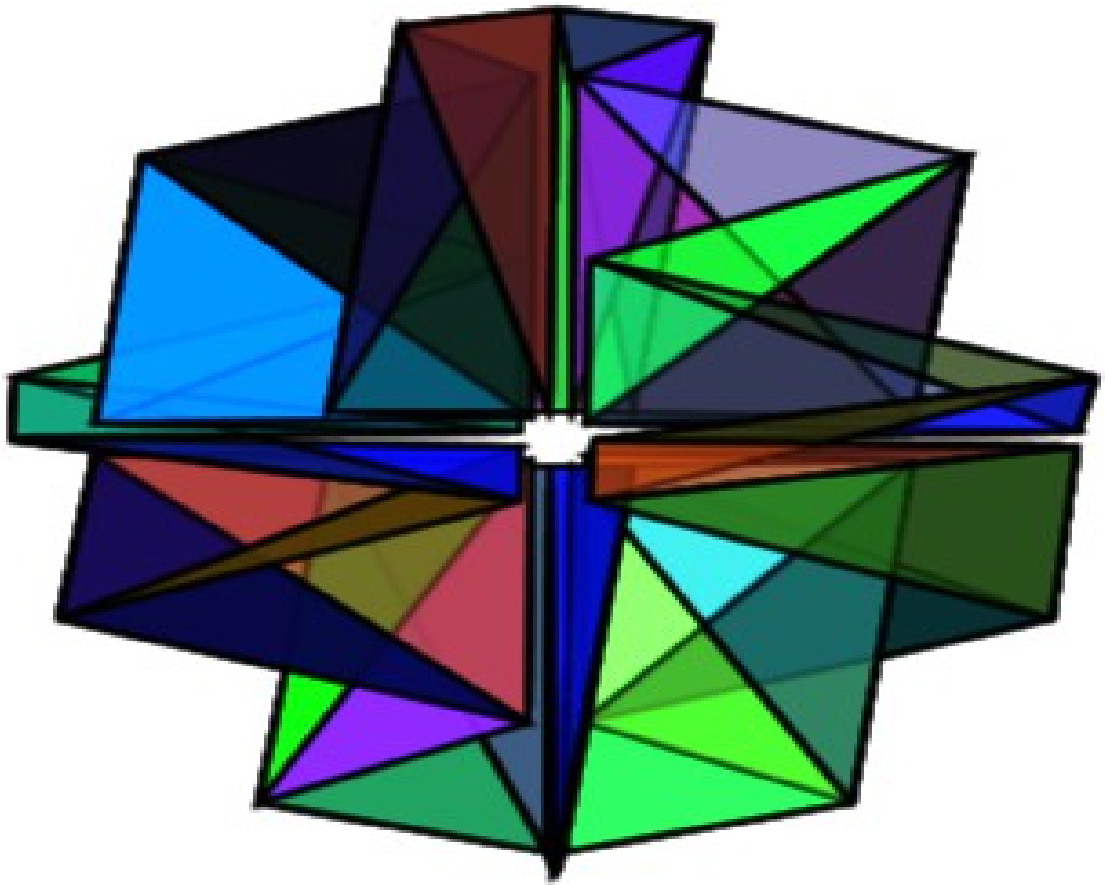


Illustration 72

```
int numPoints = int(map(313, 0, width, 6, 60));
```

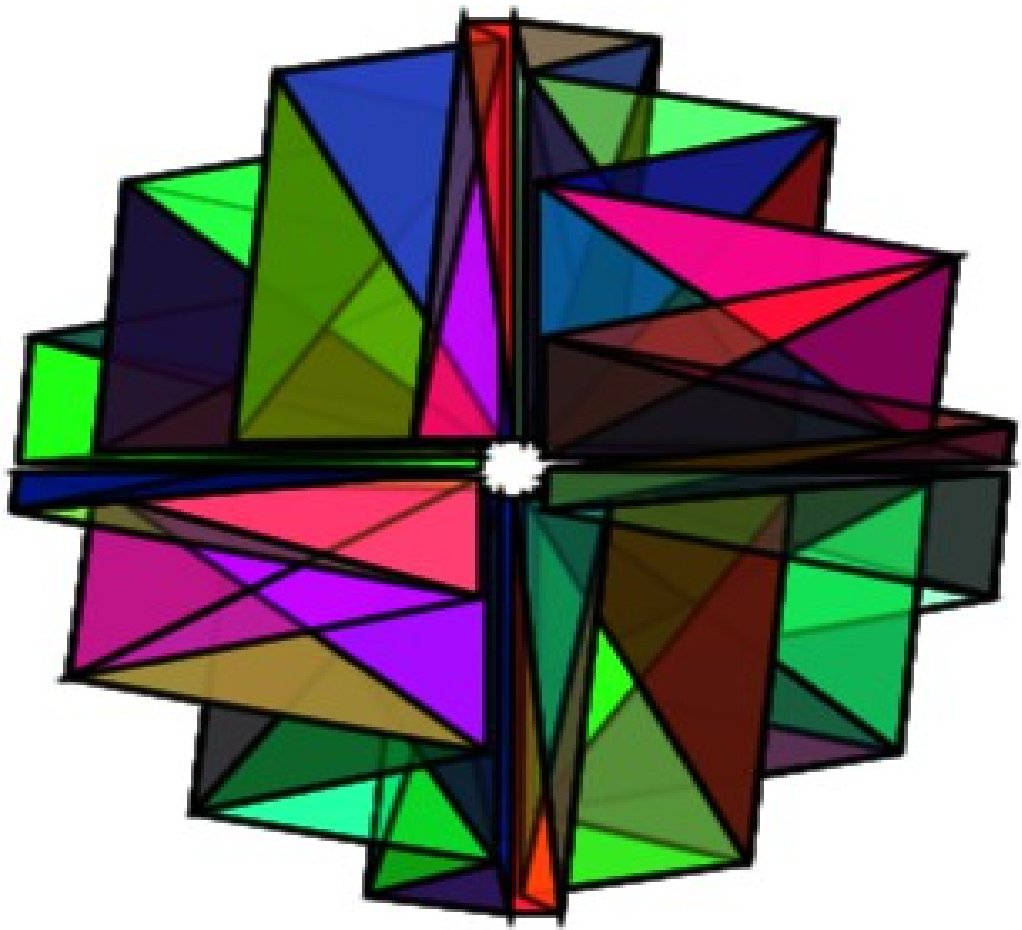


Illustration 73

```
int numPoints = int(map(527, 0, width, 6, 60));
```

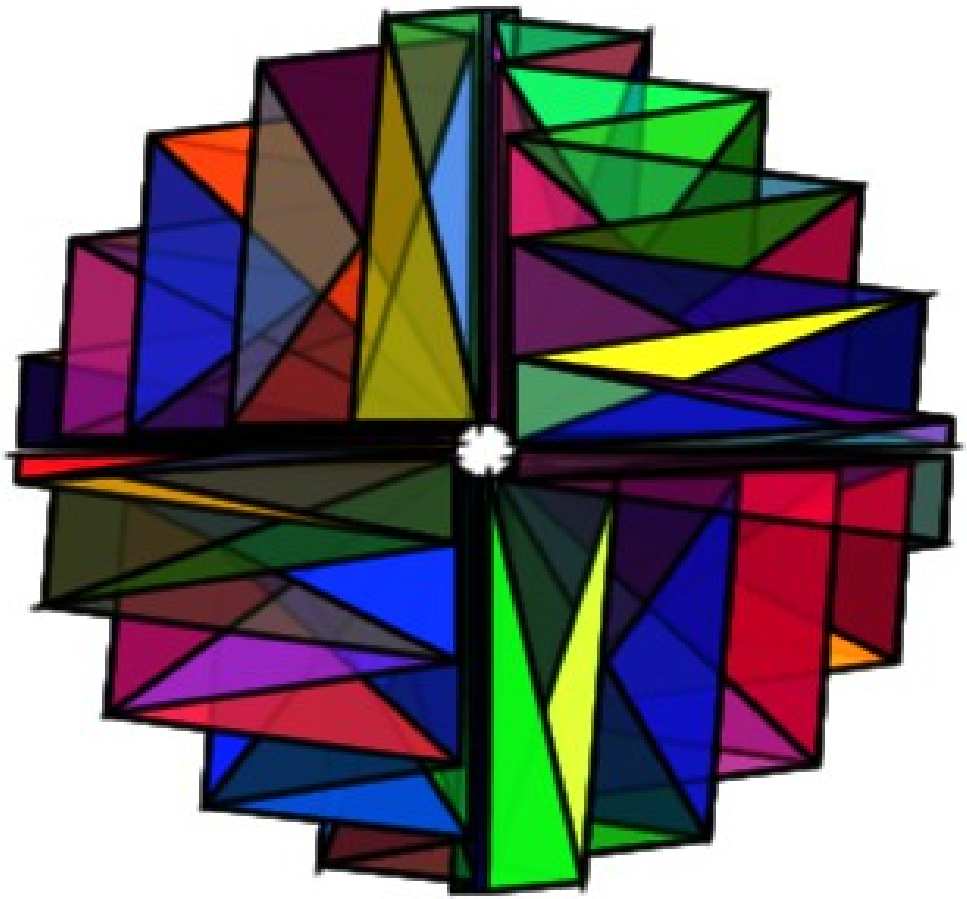


Illustration 74

```
int numPoints = int(map(427, 0, width, 6, 60));
```

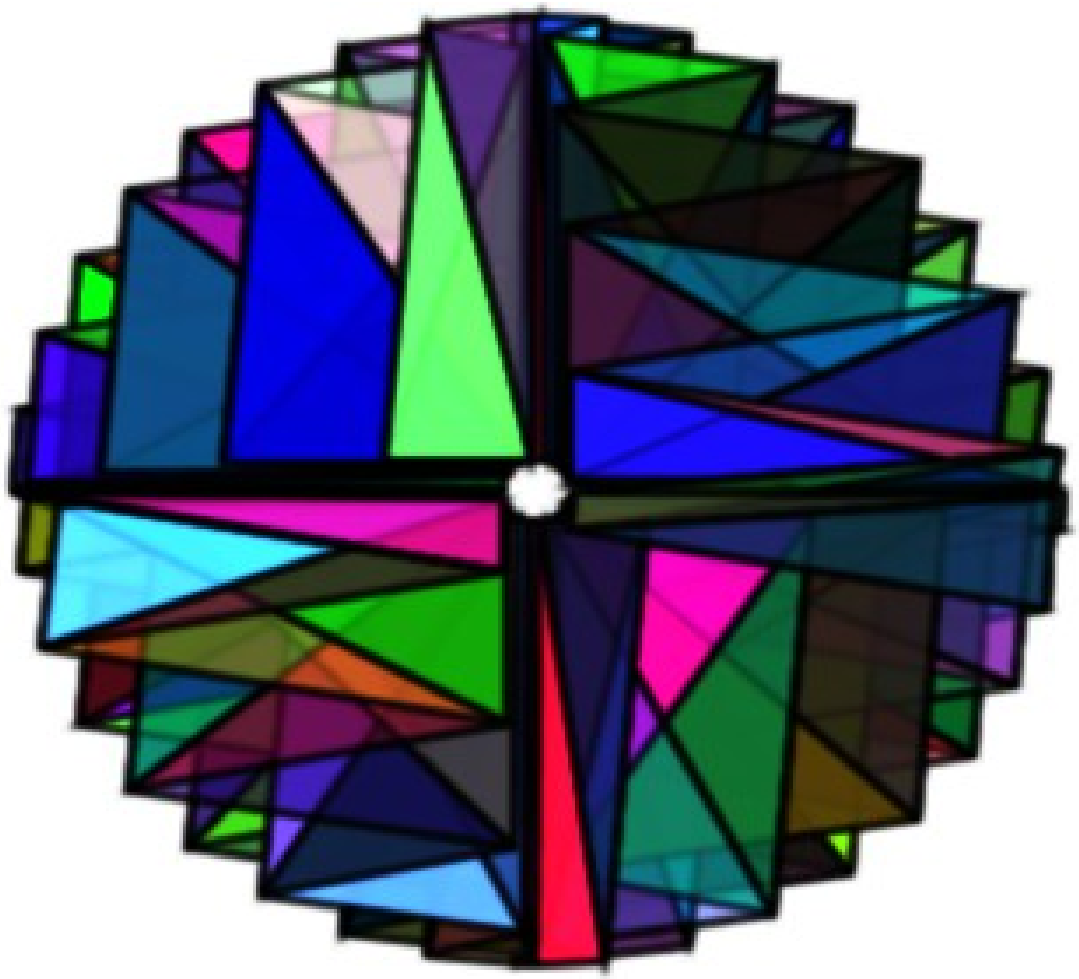


Illustration 75

Shape 25

```
beginShape(PShape.TRIANGLE_STRIP);
 for (int i = 0; i <= numPoints; i++) {
  float px =x + cos(radians(angle)) * insideRadius;
  float py = y + sin(radians(angle)) * insideRadius;
  angle += (angleStep);
  fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
  vertex(px, py);
  px = x+cos(radians(angle)) * insideRadius;
  py = y+sin(radians(angle)) * outsideRadius;
  angle += (angleStep);
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
  px = x+cos(radians(angle)) * insideRadius;
  py = y+sin(radians(angle)) * outsideRadius;
  angle += (angleStep);
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
  px = x+cos(radians(angle)) * outsideRadius;
  py = y+sin(radians(angle)) * outsideRadius;
  angle += (angleStep);
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
 }
 endShape();
```

```
int numPoints = int(map(337, 0, width, 6, 60));
```

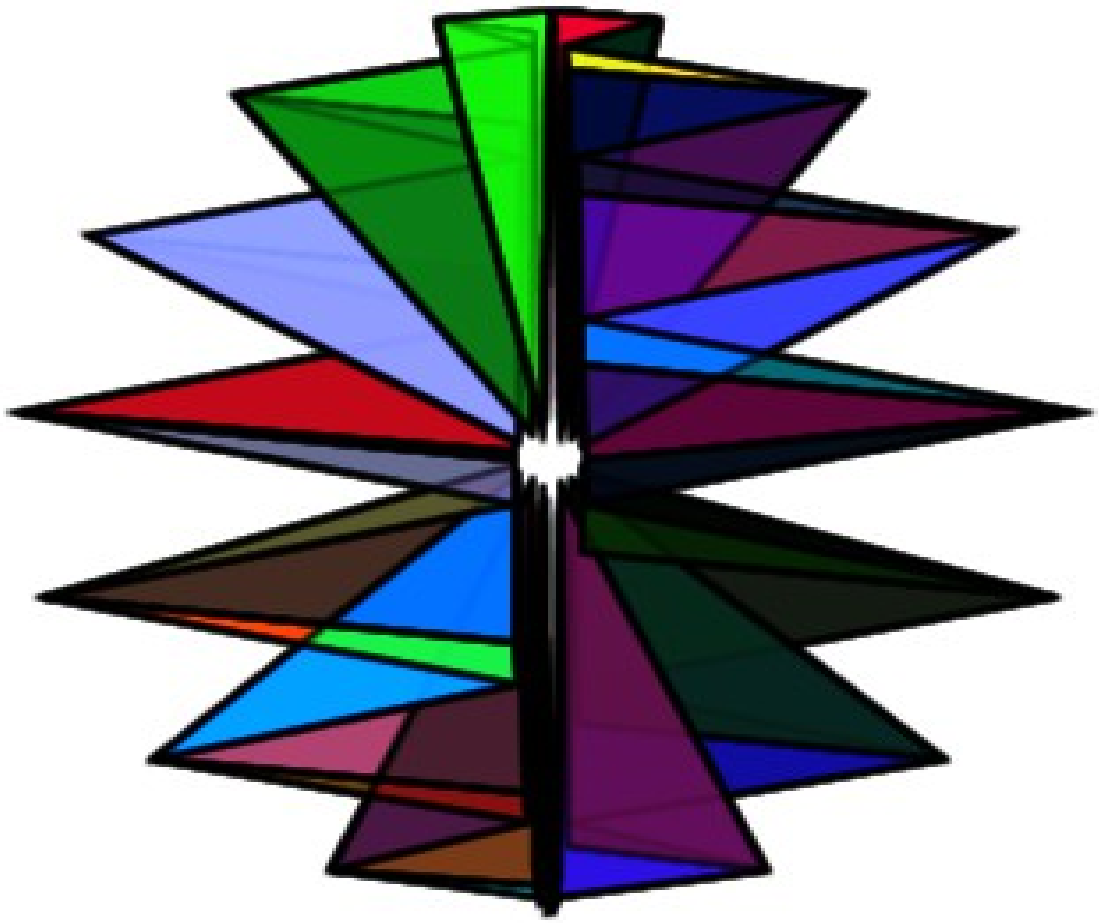


Illustration 76

```
int numPoints = int(map(582, 0, width, 6, 60));
```

Illustration 77

Shape 26

```
beginShape(PShape.TRIANGLE_STRIP);
 for(int i = 0; i <= numPoints; i++) {
   float px =x + cos(radians(angle)) * outsideRadius;
   float py = y + sin(radians(angle)) * outsideRadius;
   angle += (angleStep);
   fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
   vertex(px, py*1.25);
   px = x+cos(radians(angle)) * insideRadius;
   py = y+sin(radians(angle)) * outsideRadius;
   angle += (angleStep);
   fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
   vertex(px, py*1.25);
   px = x+cos(radians(angle)) * insideRadius;
   py = y+sin(radians(angle)) * insideRadius;
   angle += (angleStep);
   fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
   vertex(px, py*1.25);
   px = x+cos(radians(angle)) * insideRadius;
   py = y+sin(radians(angle)) * insideRadius;
   angle += (angleStep);
   fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
   vertex(px, py*1.25);
 }
 endShape();
```

int numPoints = int(map(526, 0, width, 6, 60));

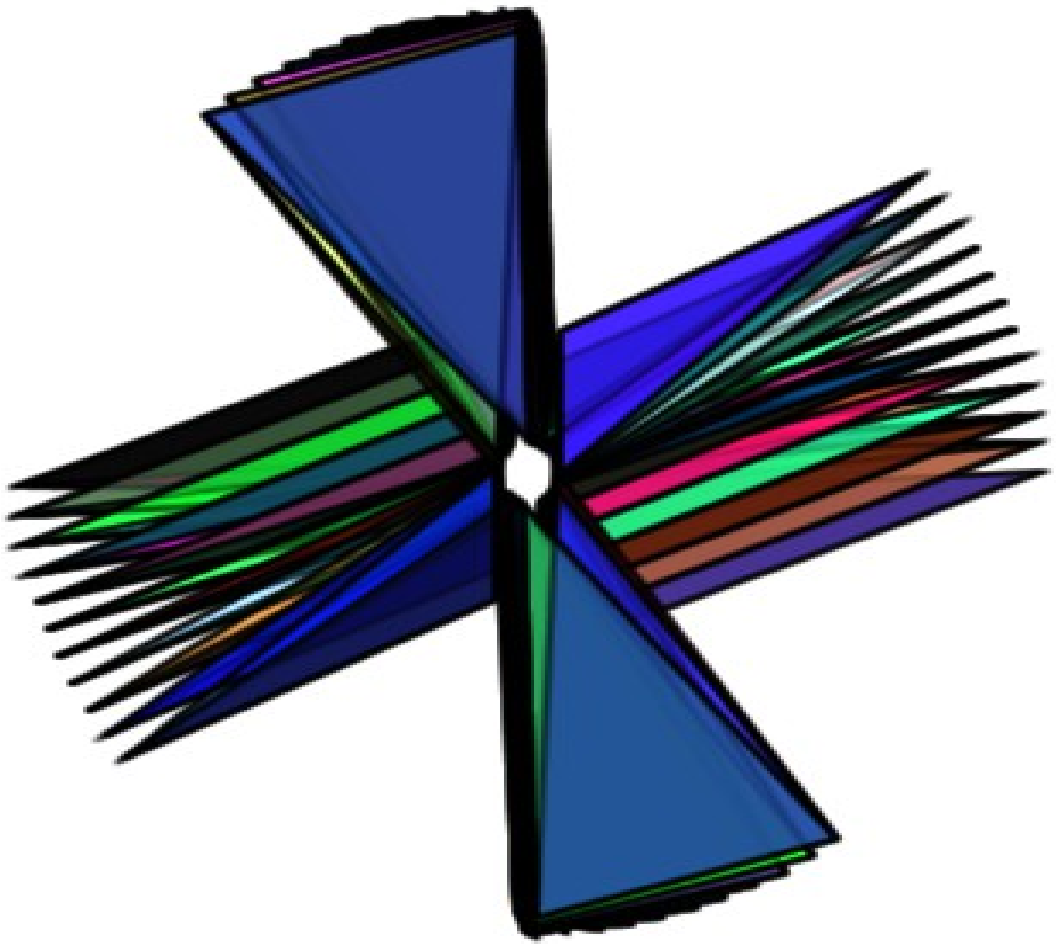


Illustration 78

```
int numPoints = int(map(526, 0, width, 6, 60));
```

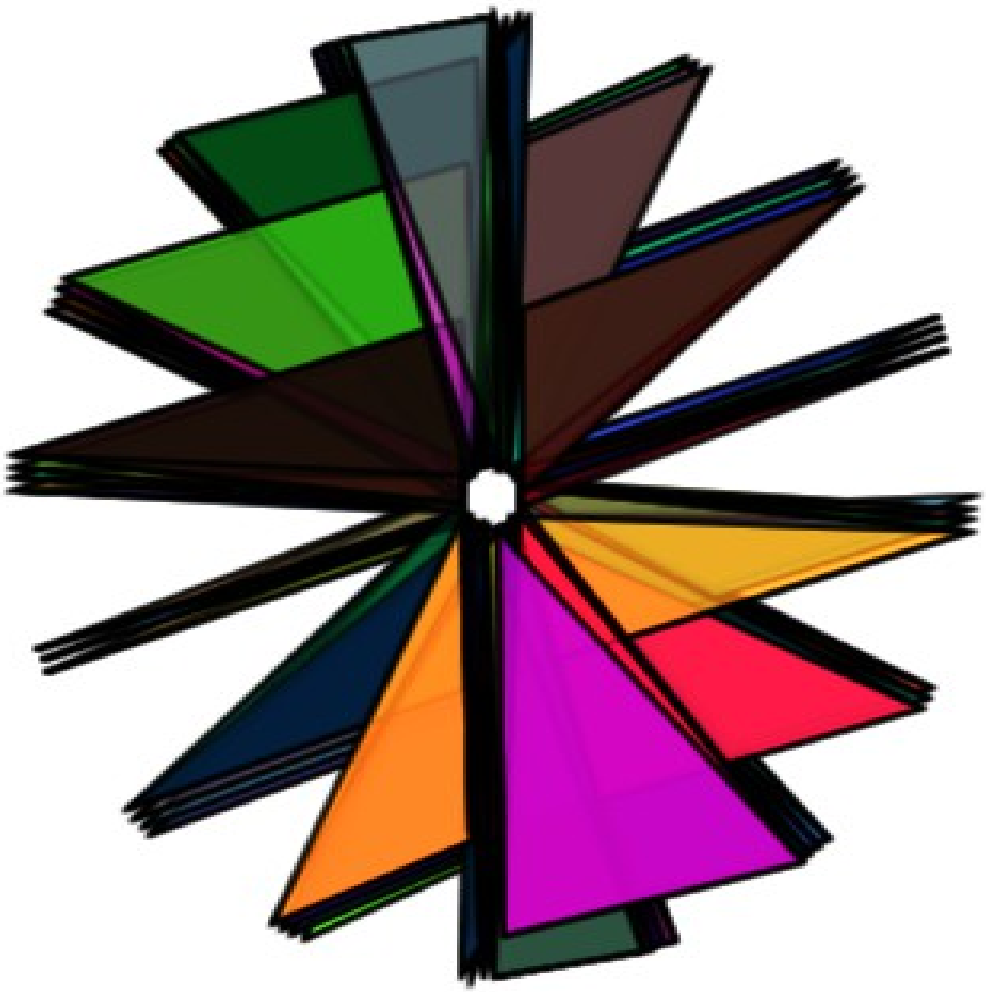


Illustration 79

```
int numPoints = int(map(368, 0, width, 6, 60));
```

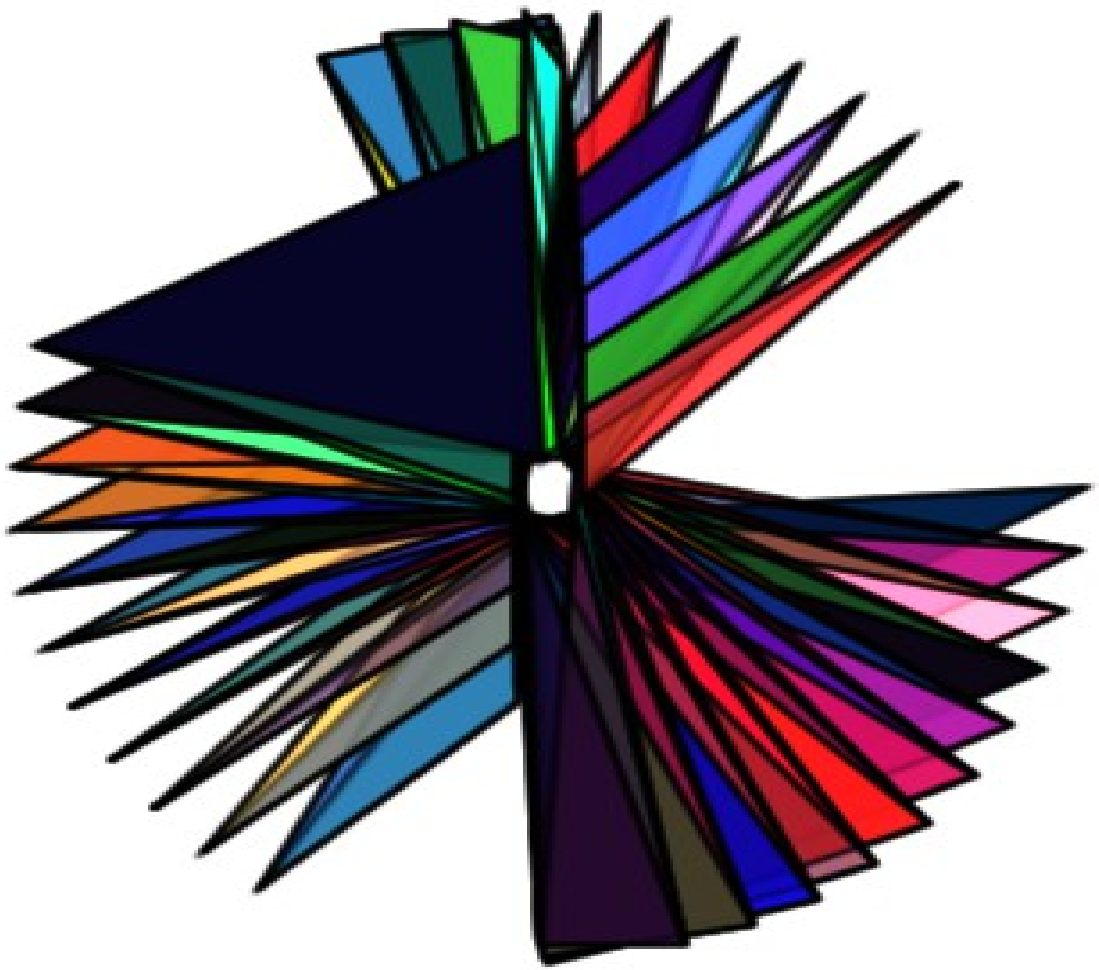


Illustration 80

```
int numPoints = int(map(368, 0, width, 6, 60));
```

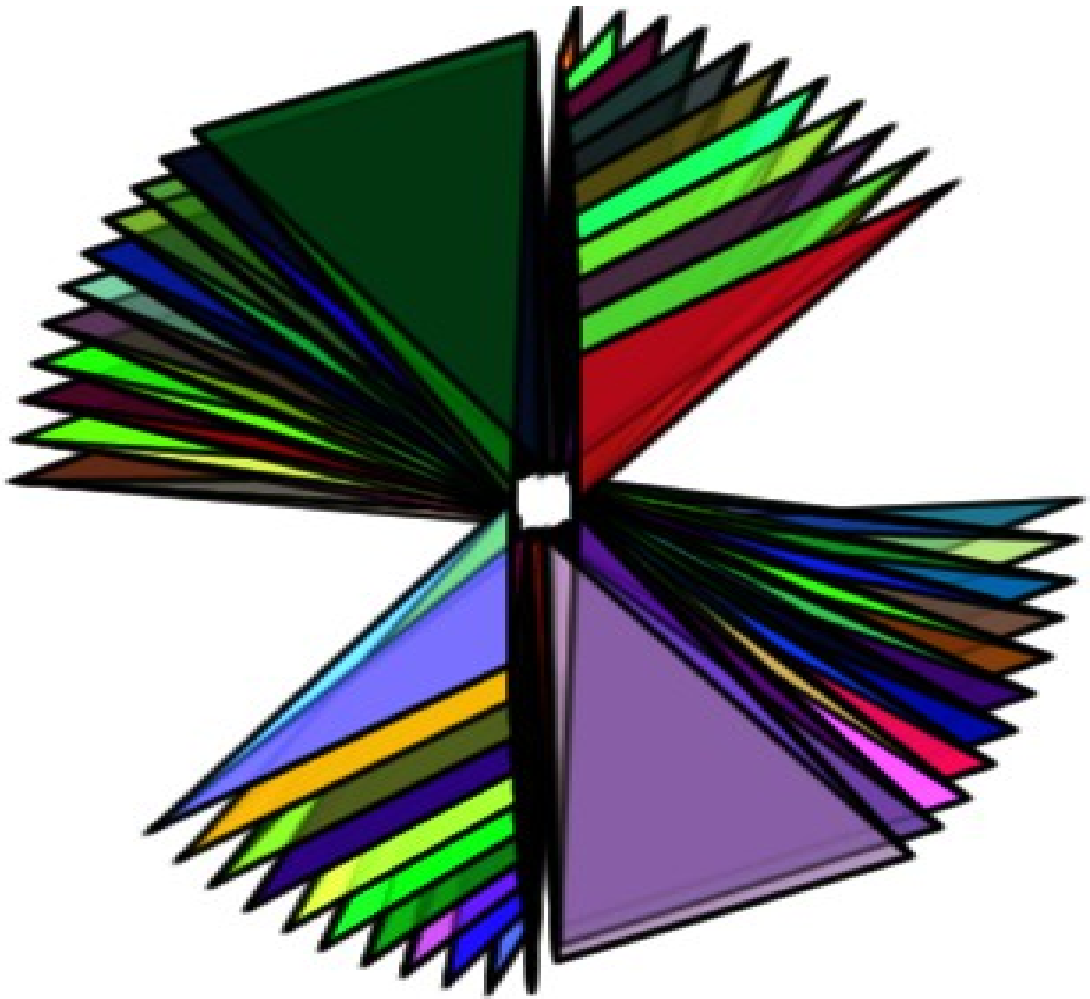


Illustration 81

```
int numPoints = int(map(445, 0, width, 6, 60));
```

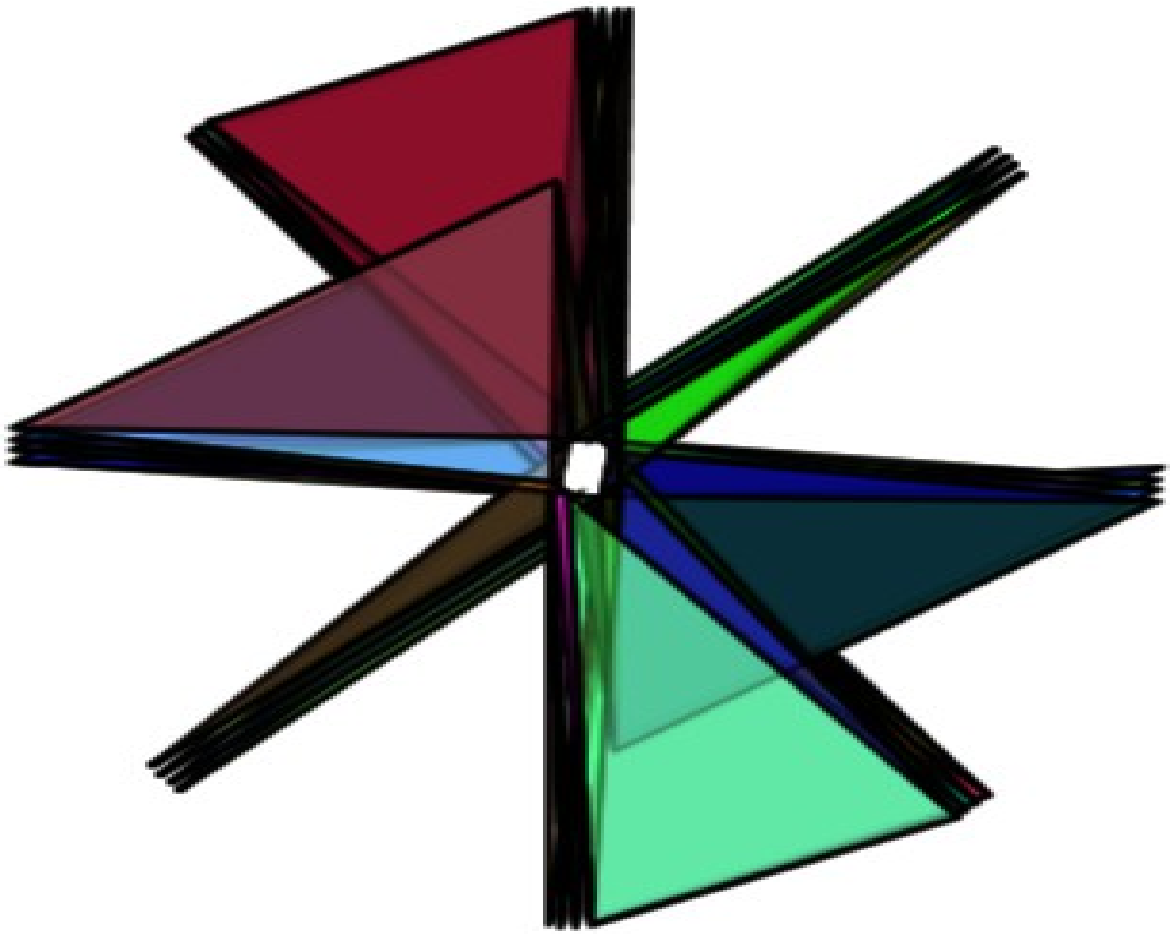


Illustration 82

Shape 27

```
beginShape(PShape.TRIANGLE_STRIP);
for (int i = 0; i <= numPoints*5; i++) {
  float px =x + cos(radians(angle)) * outsideRadius;
  float py = y + sin(radians(angle)) * outsideRadius;
  angle += (angleStep);
  fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
  vertex(px, py);
  px = x+cos(radians(angle)) * insideRadius;
  py = y+sin(radians(angle)) * outsideRadius;
  angle += (angleStep);
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py*1.25);
  px = x+cos(radians(angle)) * insideRadius;
  py = y+sin(radians(angle)) * insideRadius;
  angle += (angleStep);
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py*1.5);
  px = x+cos(radians(angle)) * insideRadius;
  py = y+sin(radians(angle)) * insideRadius;
  angle += (angleStep);
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py*1.8);
```

```
px = x+cos(radians(angle)) * insideRadius;
  py = y+sin(radians(angle)) * insideRadius;
  angle += (angleStep);
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px*1.3, py*1.155);
   px = x+cos(radians(angle)) * insideRadius;
  py = y+sin(radians(angle)) * insideRadius;
  angle += (angleStep);
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px*1.4, py*1.2);
   px = x+cos(radians(angle)) * insideRadius;
  py = y+sin(radians(angle)) * insideRadius;
  angle += (angleStep);
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px*1.5, py*1.4);
 }
 endShape();
```

```
int numPoints = int(map(436, 0, width, 6, 60));
```

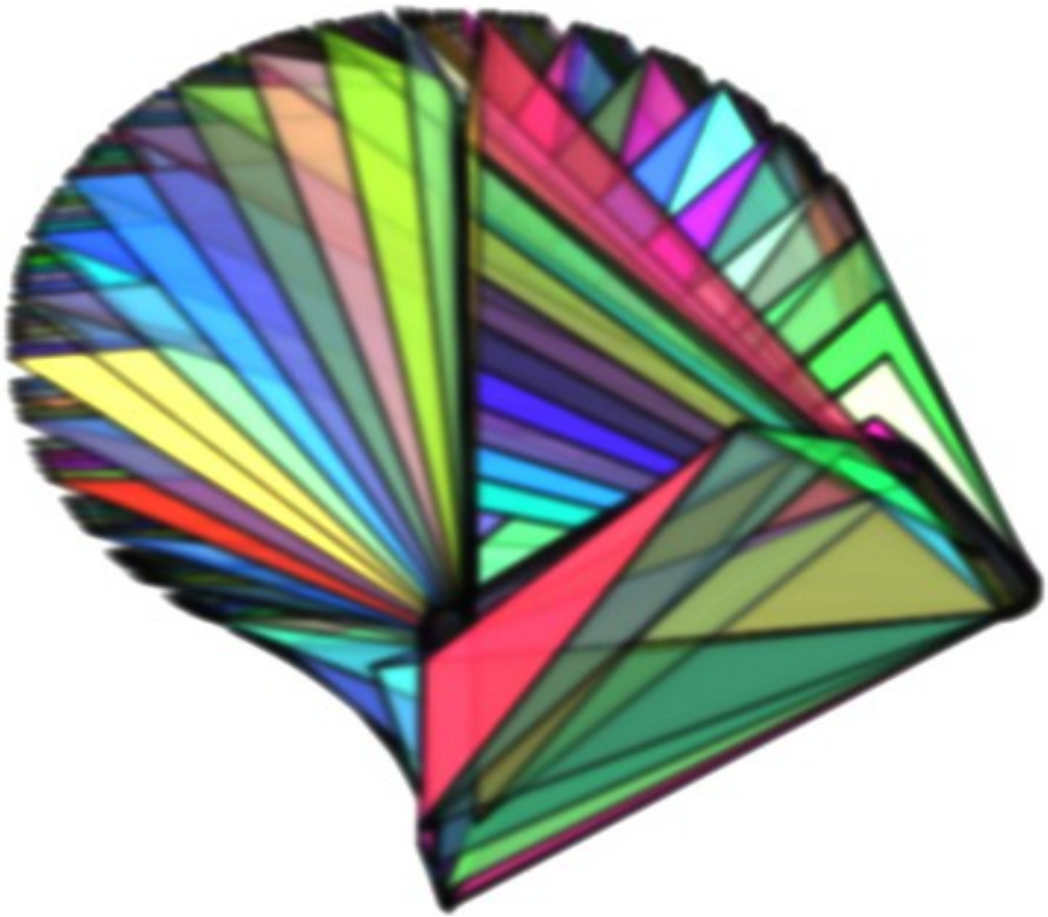


Illustration 83

```
int numPoints = int(map(459, 0, width, 6, 60));
```

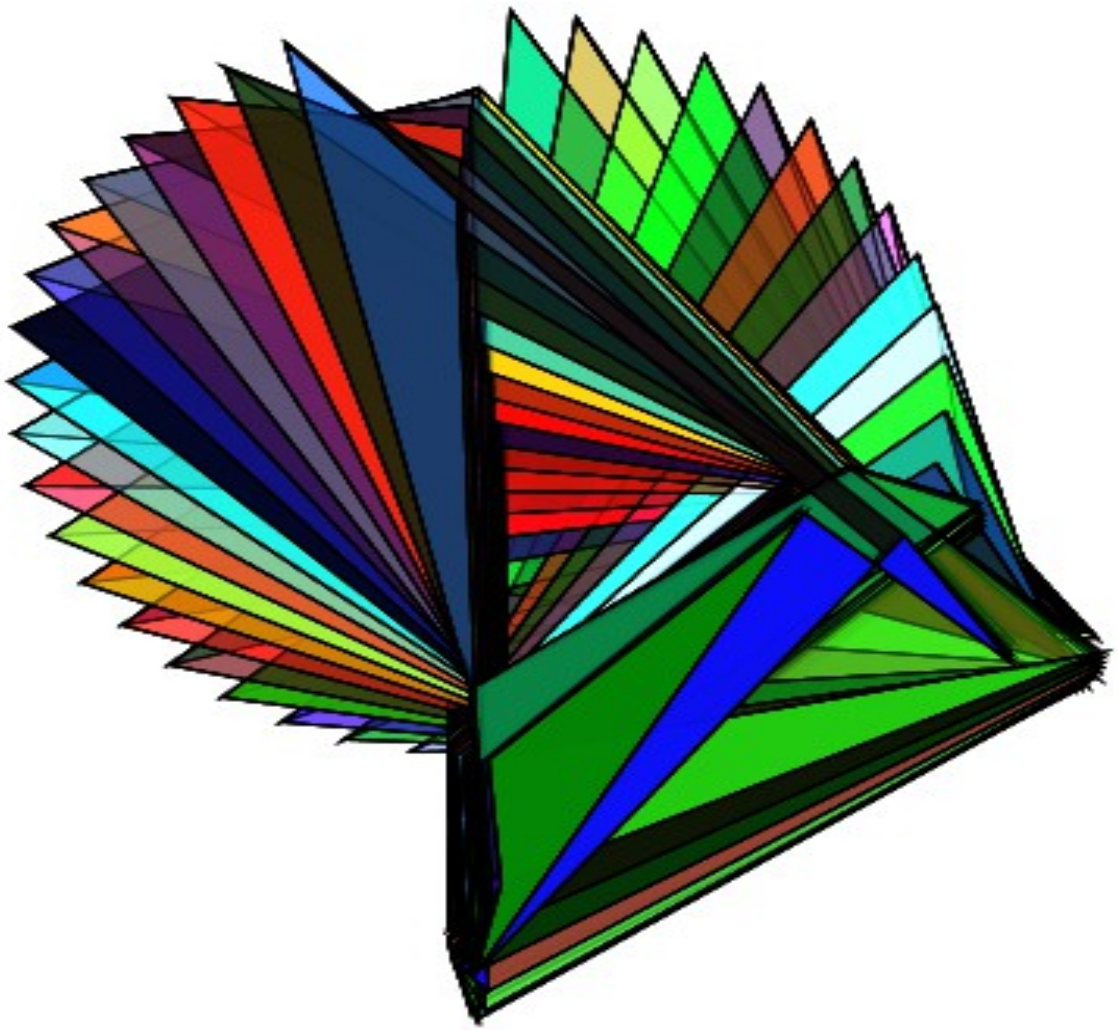


Illustration 84

```
int numPoints = int(map(705, 0, width, 6, 60));
```

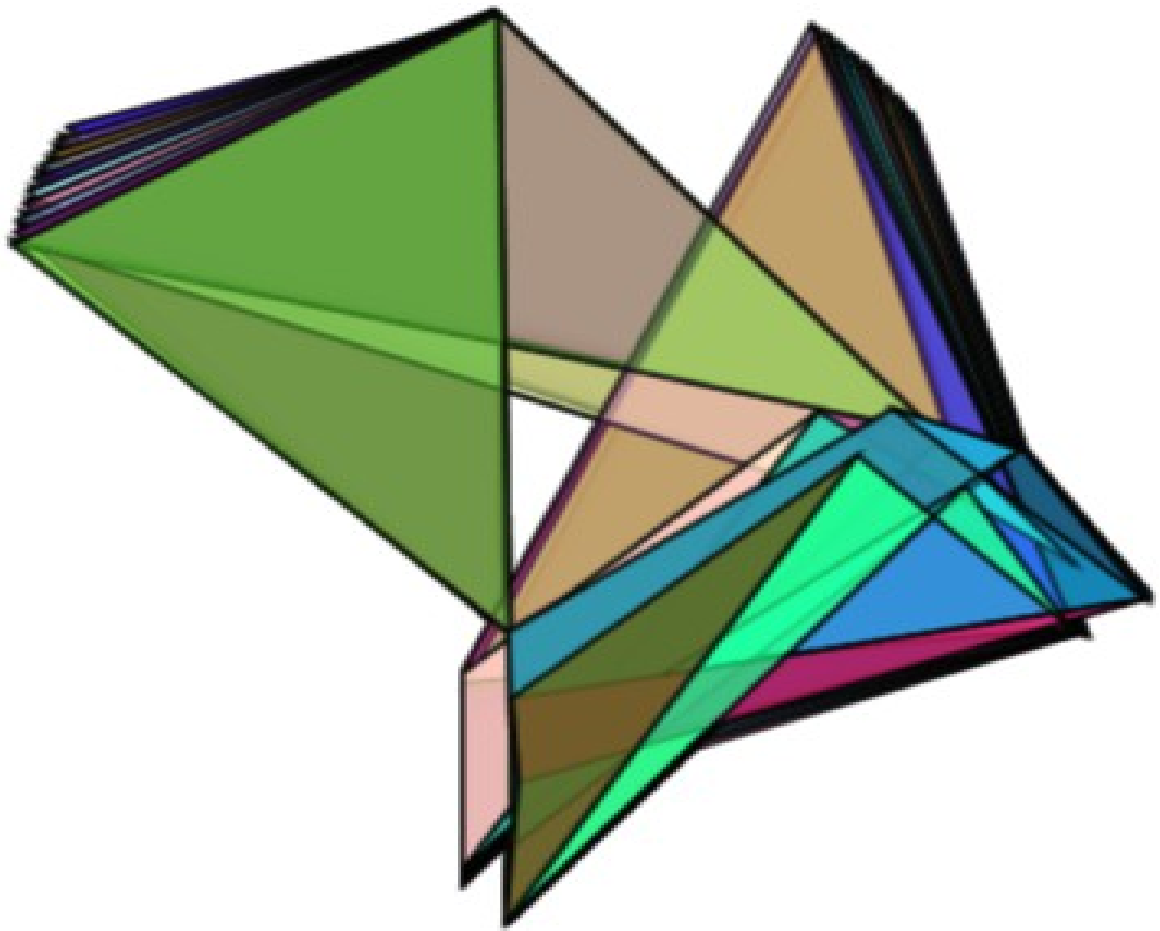


Illustration 85

```
int numPoints = int(map(450, 0, width, 6, 60));
```

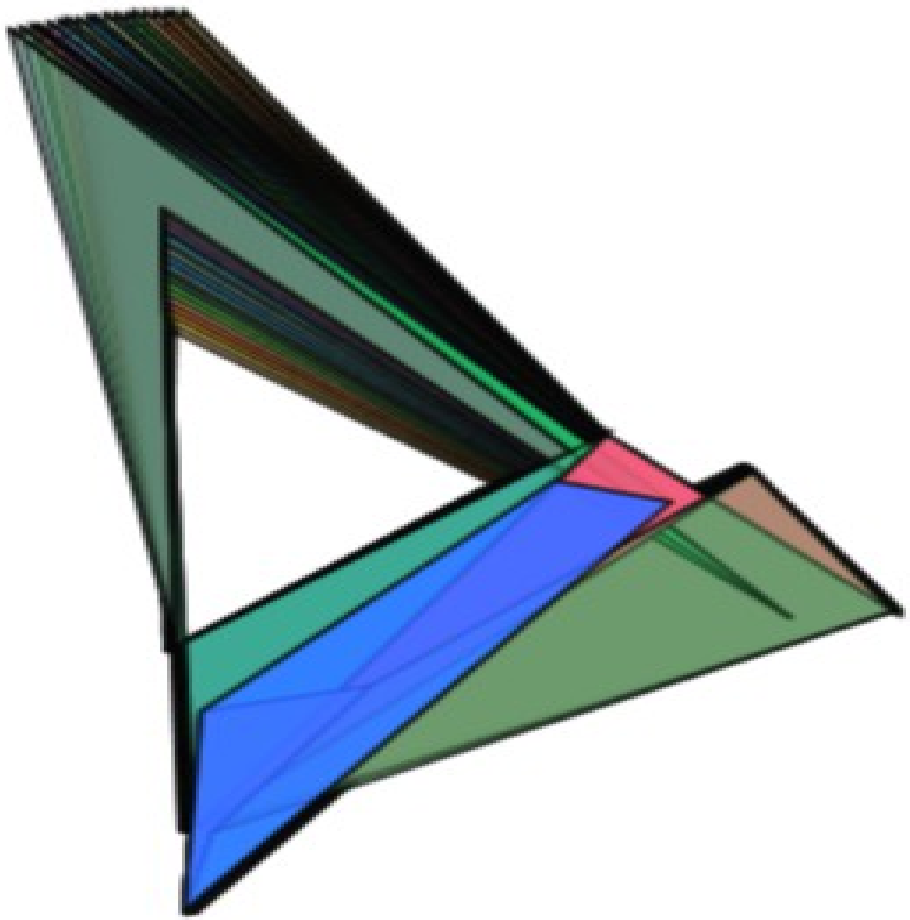


Illustration 86

Shape 28

```
  float angleStep = 980.0/numPoints;
beginShape(PShape.QUAD_STRIP);
 for (int i = 0; i <= numPoints; i++) {
   float px =x + sin(radians(angle)) * outsideRadius;
   float py = y + cos(radians(angle)) * outsideRadius;
   angle += (angleStep);
   fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
   vertex(px, py);
   px = x+sin(radians(angle)) * insideRadius;
   py = y+cos(radians(angle)) * outsideRadius;
   angle += (angleStep);
   fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
   vertex(px, py);
 }
 endShape();
```

```
int numPoints = int(map(727, 0, width, 6, 60));
```

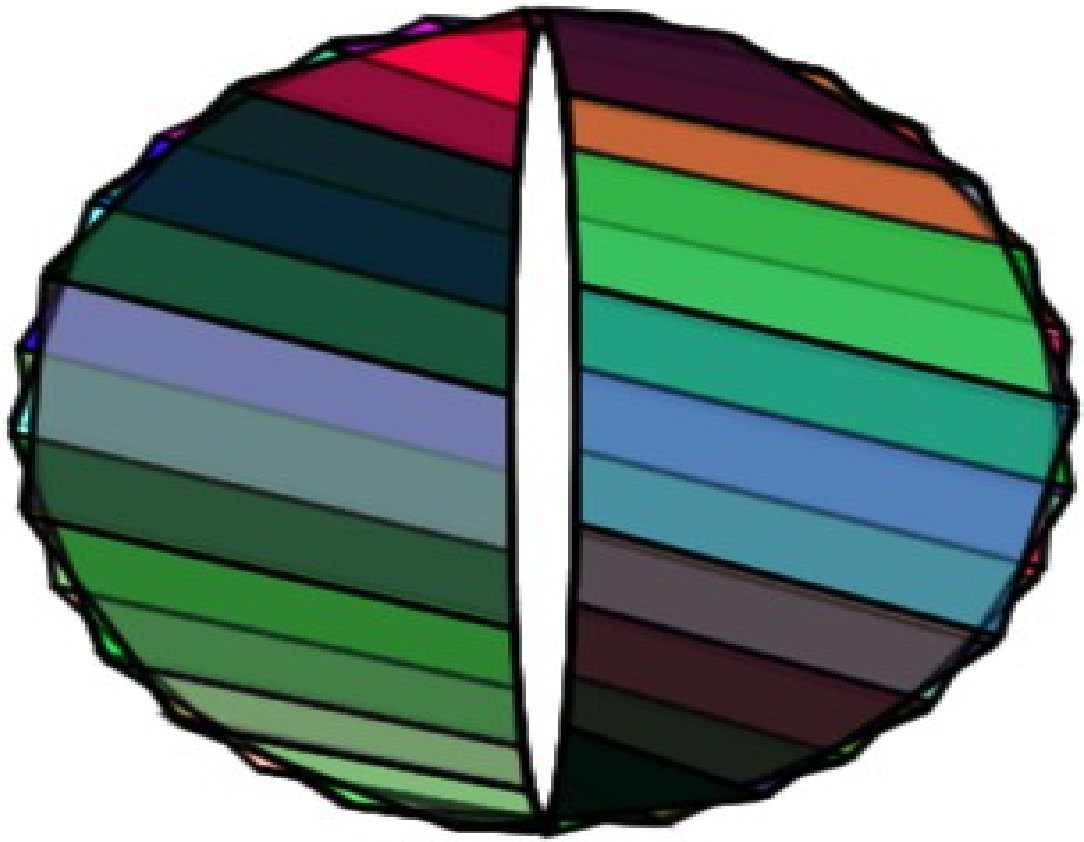


Illustration 87

```
int numPoints = int(map(340, 0, width, 6, 60));
```

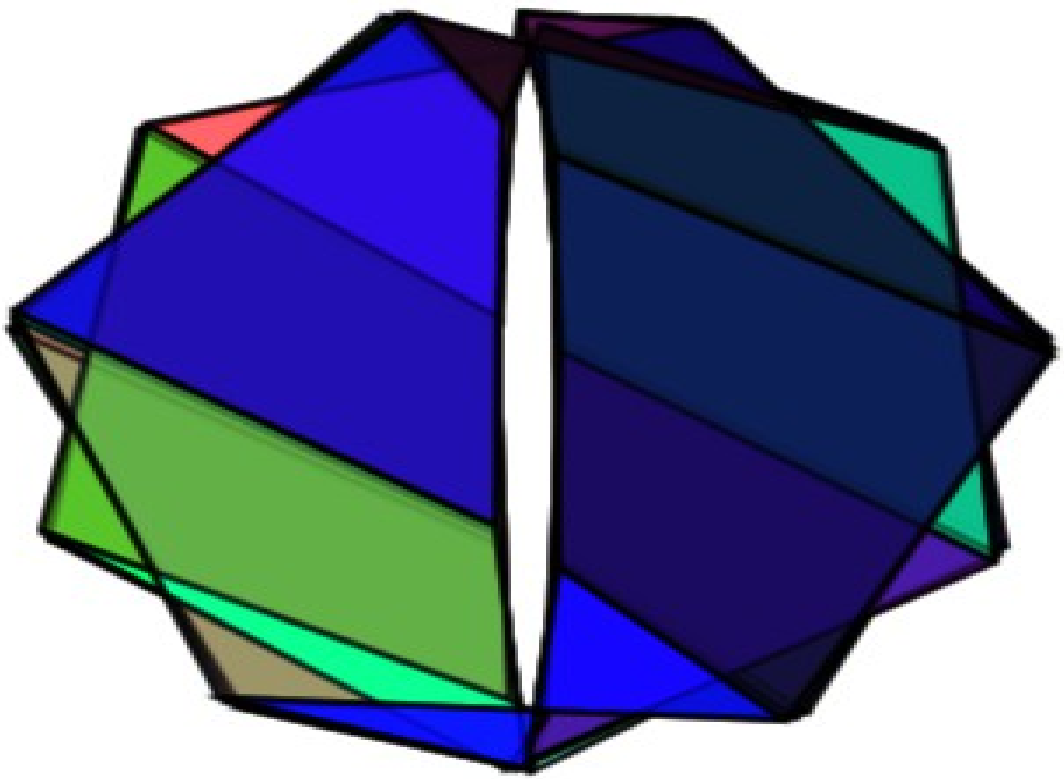


Illustration 88

```
int numPoints = int(map(312, 0, width, 6, 60));
```

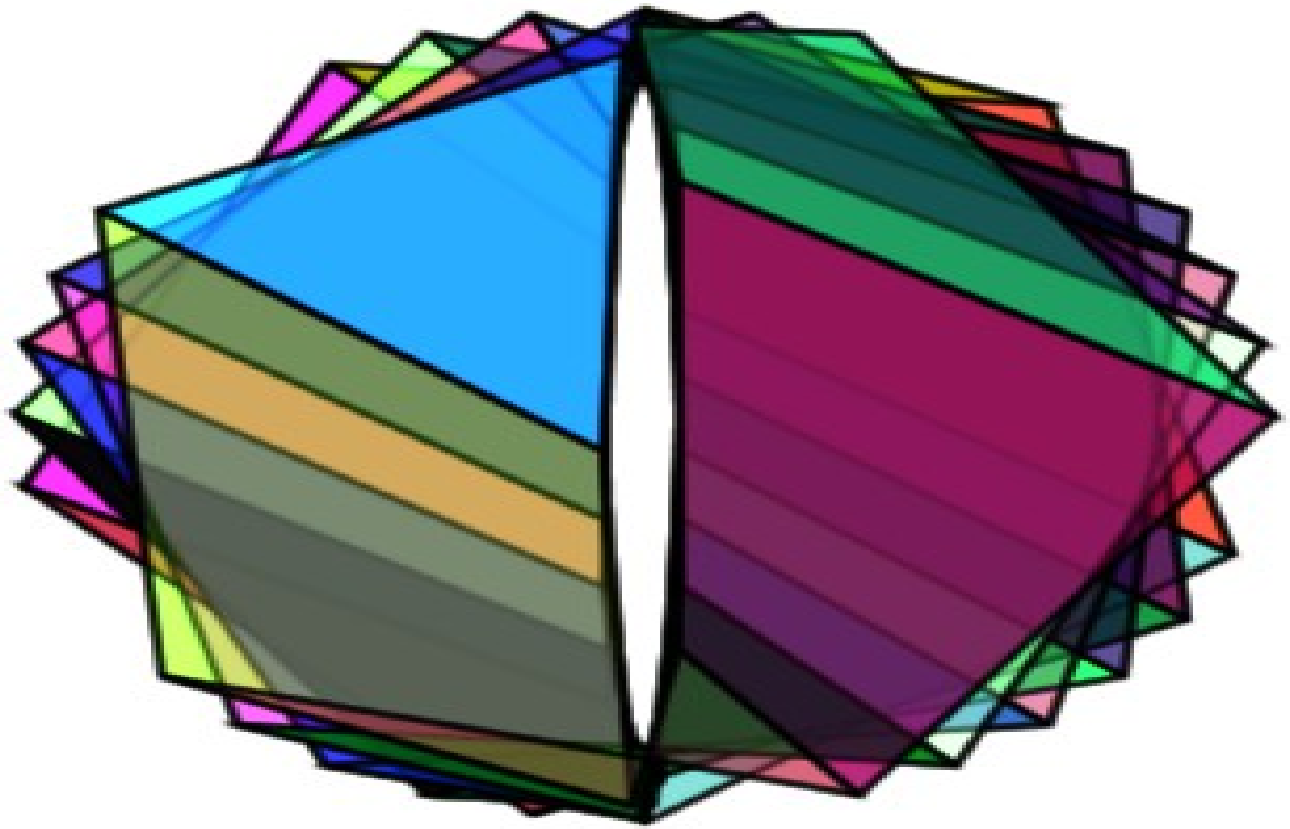


Illustration 89

```
int numPoints = int(map(222, 0, width, 6, 60));
```

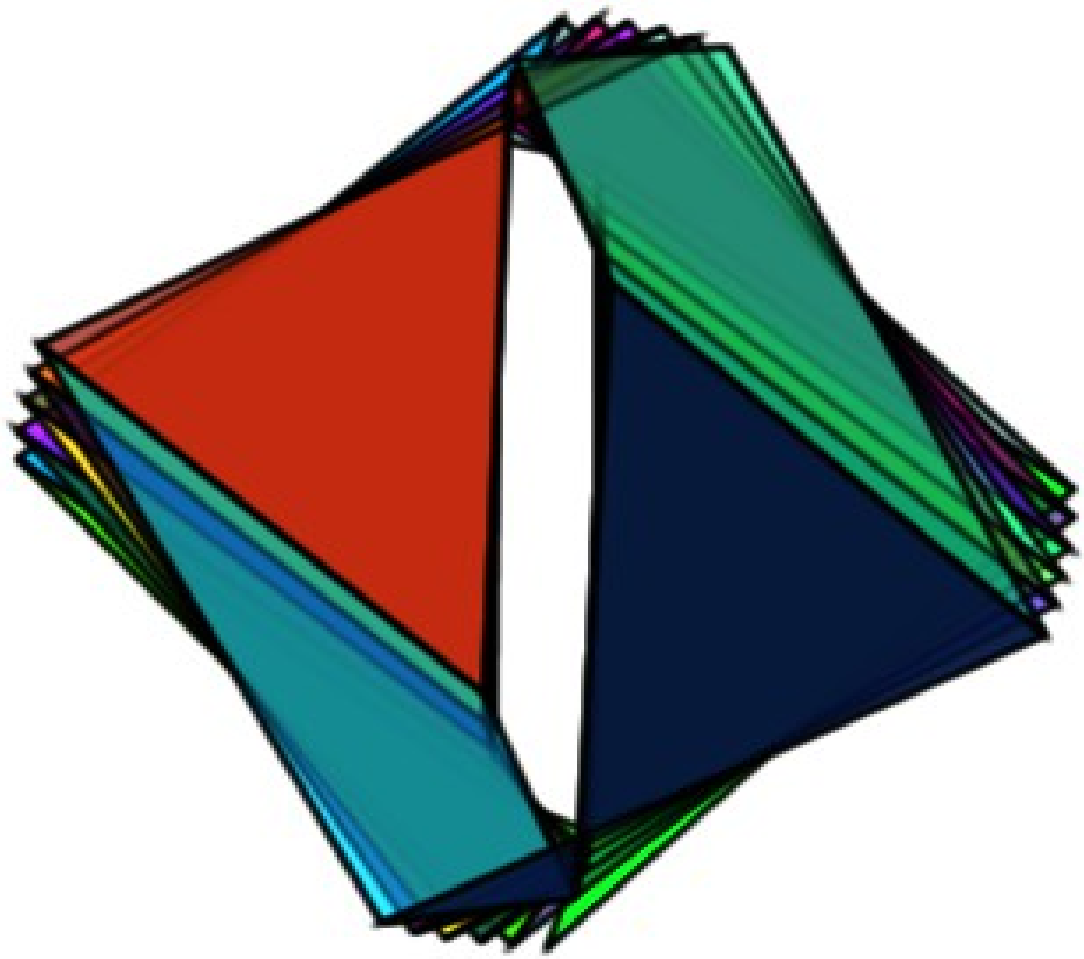


Illustration 90

Shape 29

```
beginShape(PShape.QUAD_STRIP);
 for (int i = 0; i <= numPoints; i++) {
  float px =x + sin(radians(angle)) * outsideRadius;
  float py = y + cos(radians(angle)) * outsideRadius;
  angle += (angleStep);
  fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
  vertex(px, py);
  px = x+sin(radians(angle)) * insideRadius;
  py = y+cos(radians(angle)) * outsideRadius;
  angle += (angleStep);
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
  px = x+sin(radians(angle)) * insideRadius;
  py = y+cos(radians(angle)) * outsideRadius;
  angle += (angleStep);
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
  px = x+sin(radians(angle)) * insideRadius;
  py = y+cos(radians(angle)) * outsideRadius;
  angle += (angleStep);
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
 }
 endShape();
```

int numPoints = int(map(223.0, width, 6, 60));

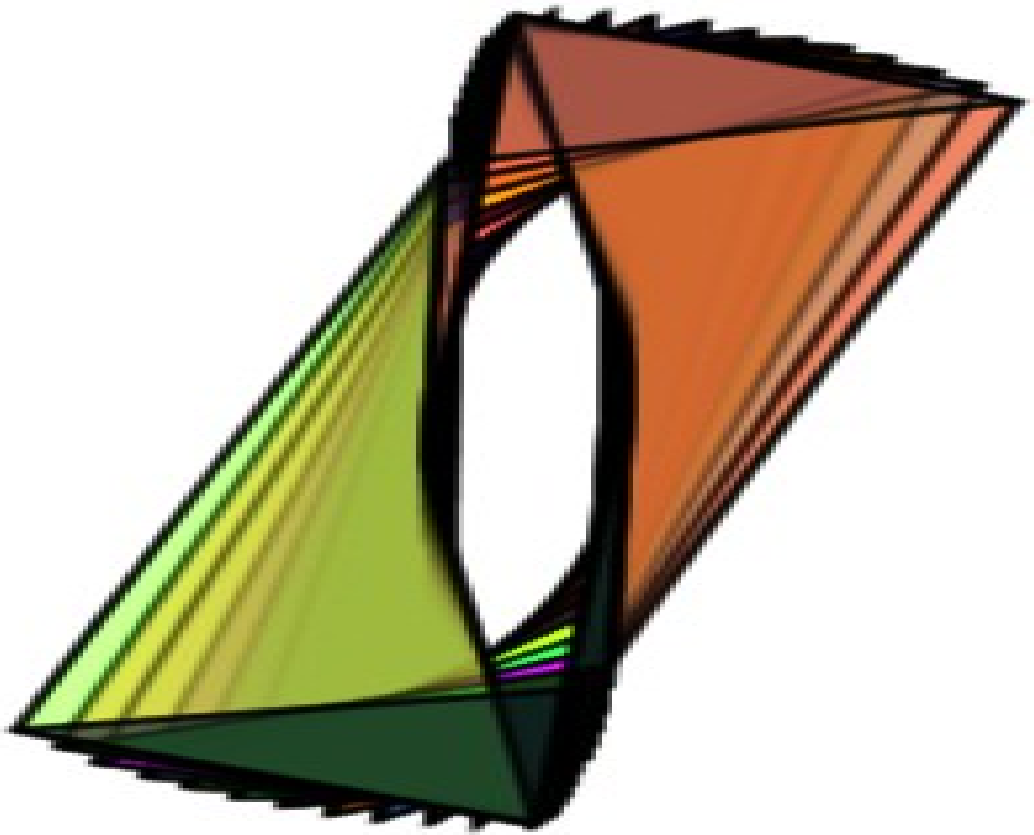


Illustration 91

```
int numPoints = int(map(357 ,width, 6, 60));
```

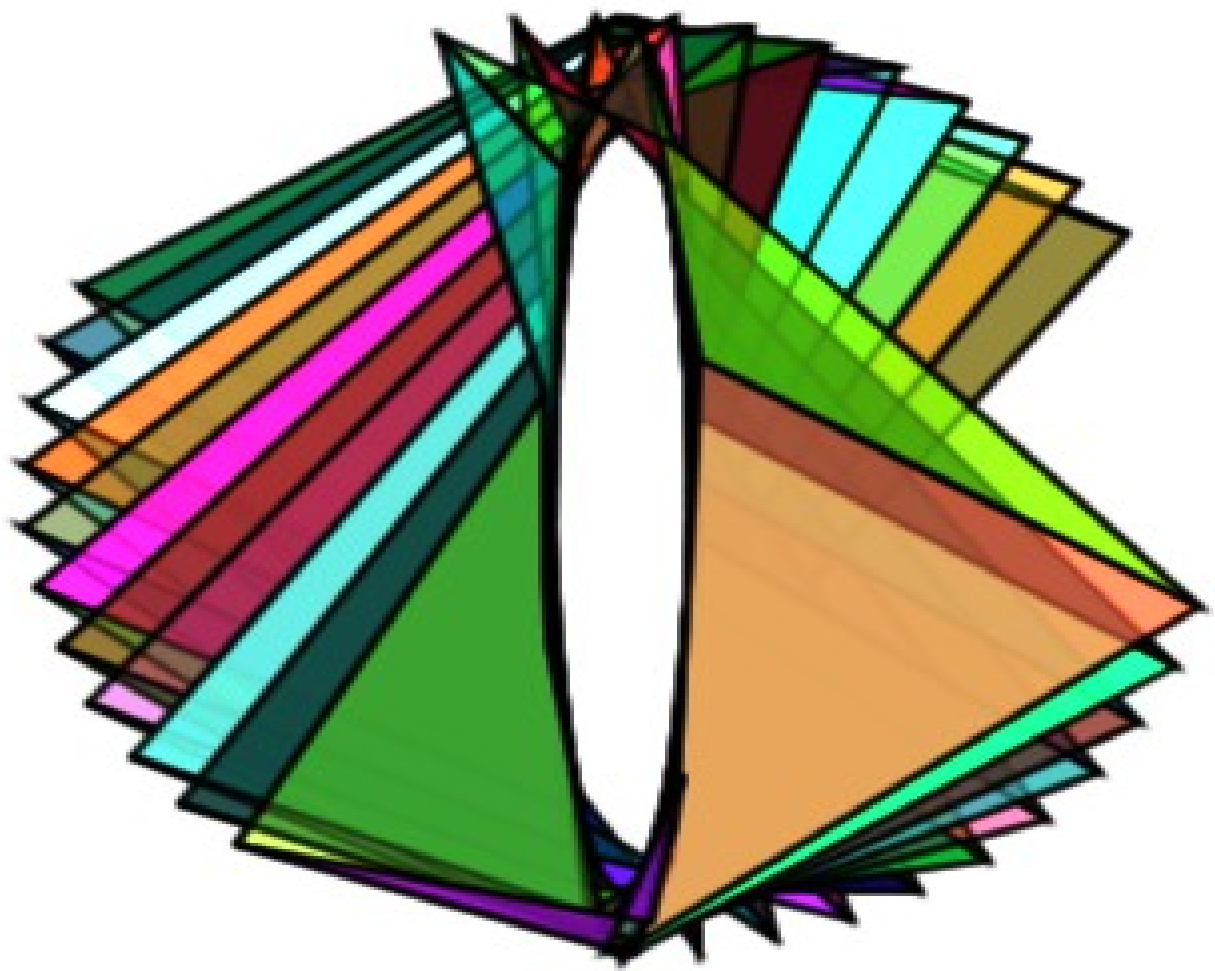


Illustration 92

```
int numPoints = int(map(668 ,width, 6, 60));
```

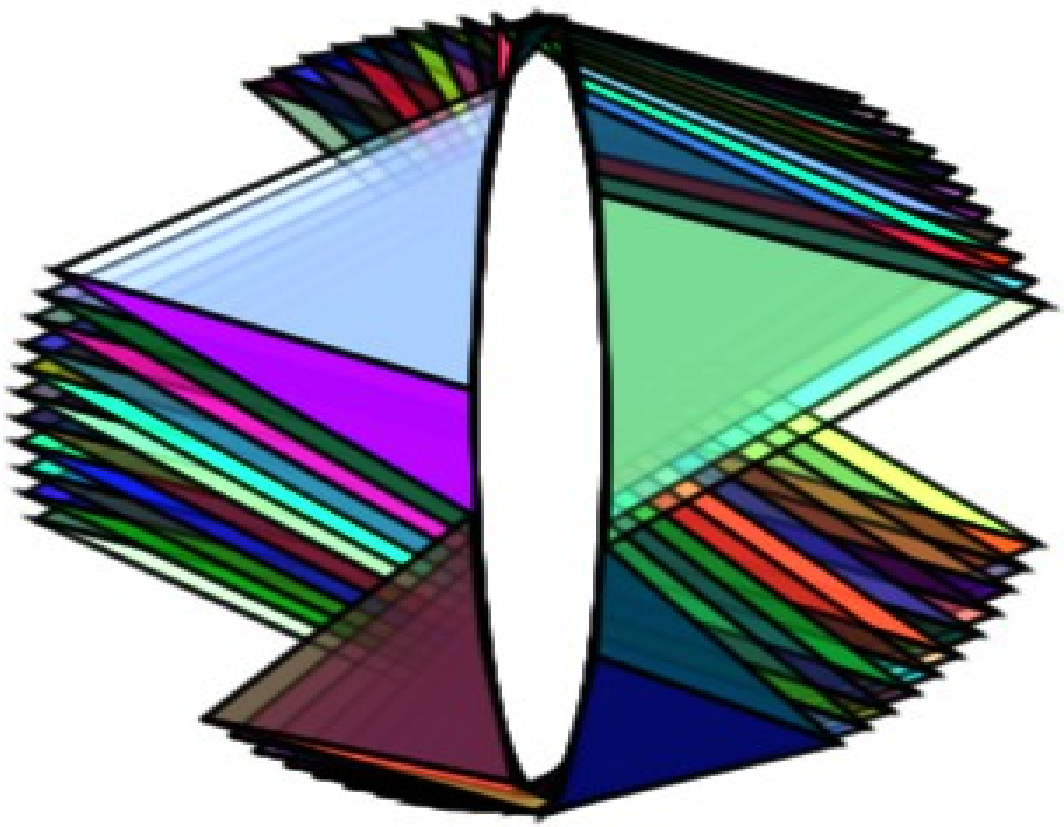


Illustration 93

```
int numPoints = int(map(595 ,width, 6, 60));
```

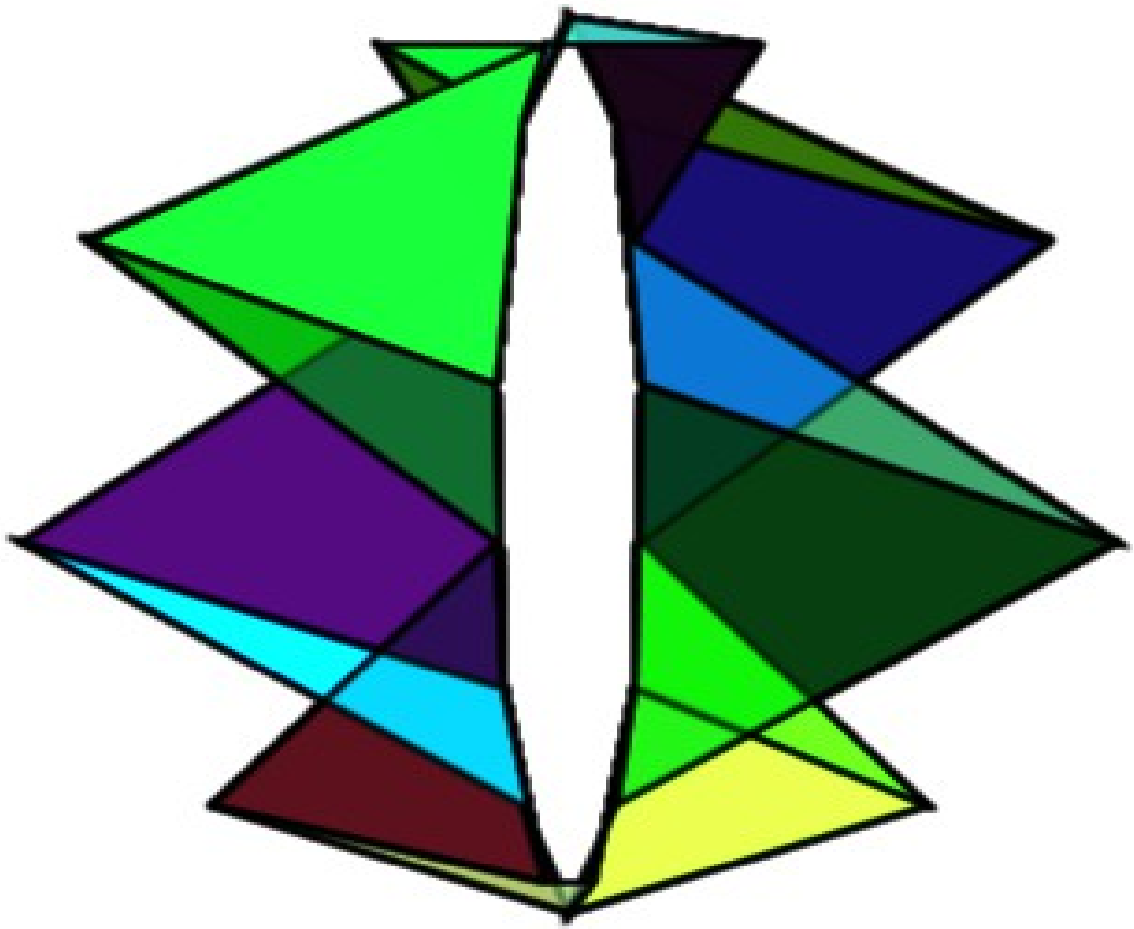


Illustration 94

Shape 30

```
beginShape(PShape.TRIANGLE_STRIP);
 for (int i = 0; i <= numPoints; i++) {
  float px = x + cos(radians(-angle)) * insideRadius;
  float py = y + sin(radians(-angle)) * insideRadius;
  angle += (angleStep);
  fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
  vertex(px, py);
  angle1 += (angleStep);
  px = x + cos(radians(-angle1)) * insideRadius;
  py = y + sin(radians(-angle1)) * outsideRadius;
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
  angle1 += (2*angleStep);
  //angle += (2*angleStep);
 }
 endShape();
```

```
int numPoints = int(map(456 ,width, 6, 60));
```

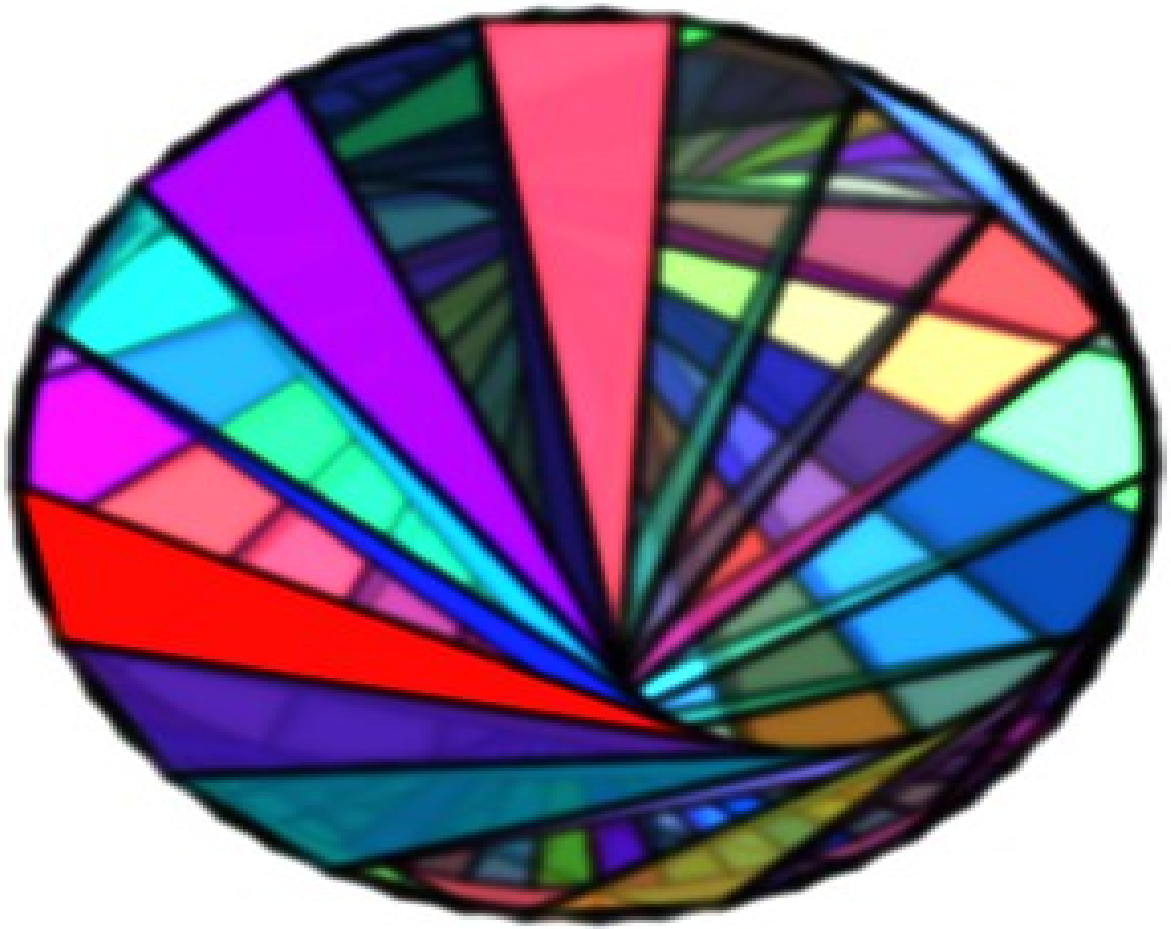


Illustration 95

```
int numPoints = int(map(450 ,width, 6, 60));
```

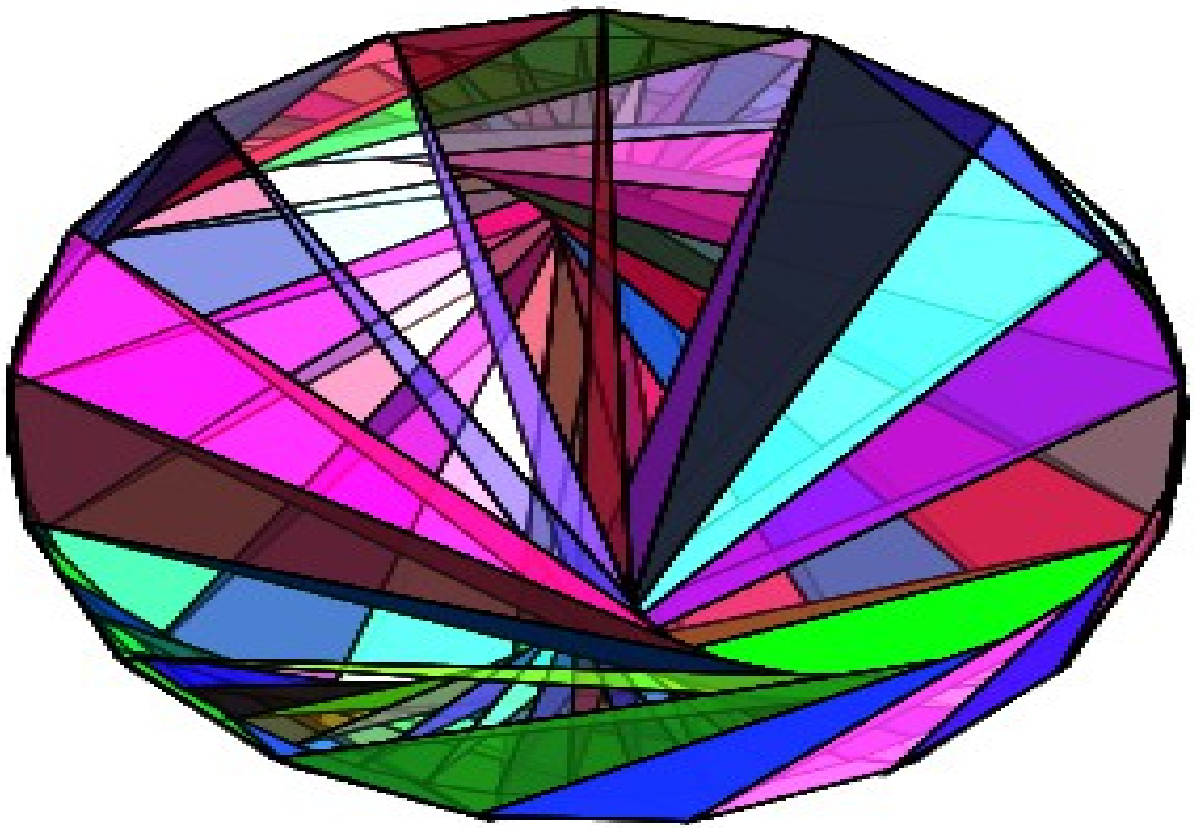


Illustration 96

Shape 31

```
beginShape(PShape.QUAD_STRIP);
for (int i = 0; i <= numPoints; i++) {
  float px =x + sin(radians(angle)) * outsideRadius;
  float py = y + cos(radians(angle)) * outsideRadius;
  angle += (angleStep);
  fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
  vertex(px, py);
  px = x+sin(radians(angle1)) * insideRadius;
  py = y+cos(radians(angle1)) * outsideRadius;
  angle1 += (angleStep);
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
  px = x+sin(radians(angle1-angle)) * insideRadius;
  py = y+cos(radians(angle1-angle)) * outsideRadius;
  angle += (angleStep);
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
  px = x+sin(radians(angle1)) * insideRadius;
  py = y+cos(radians(angle1)) * outsideRadius;
  angle1 += (angleStep);
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
}
endShape();
```

```
int numPoints = int(map(450 ,width, 6, 60));
```

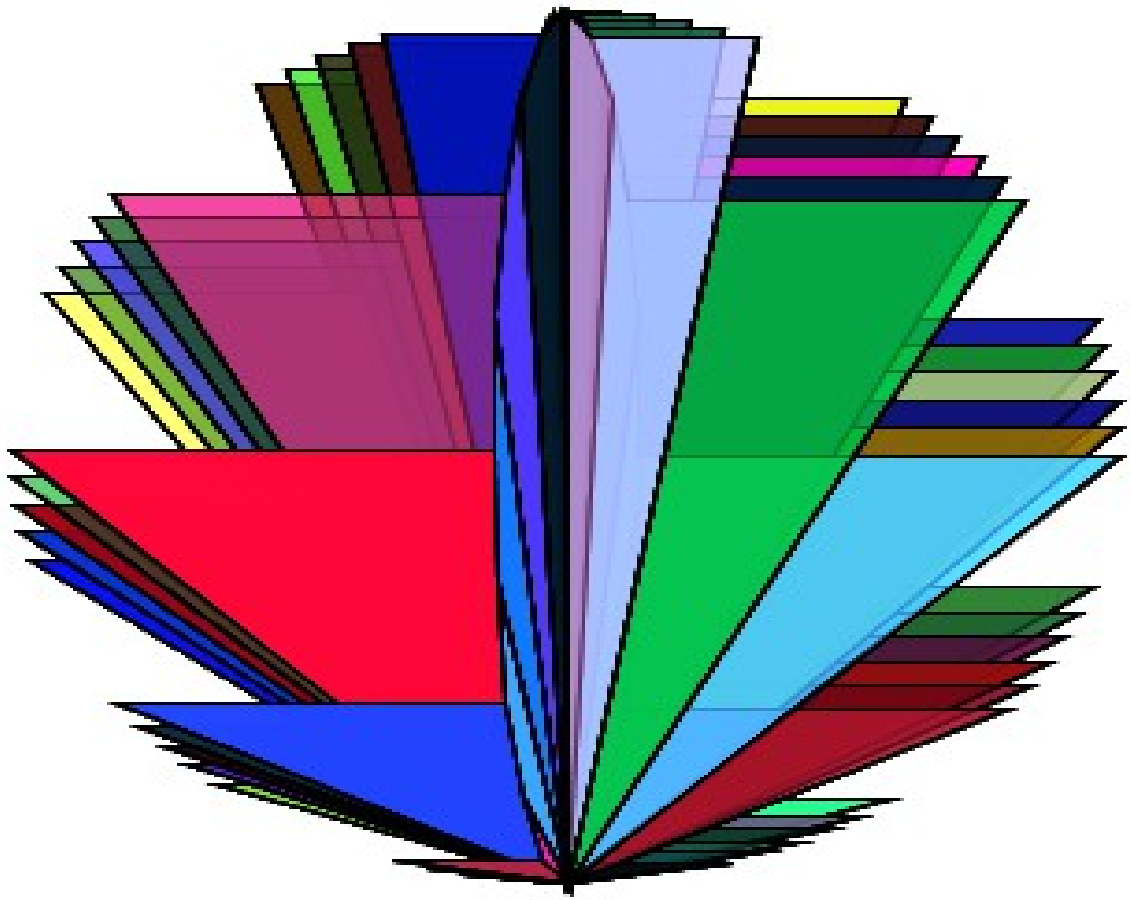


Illustration 97

int numPoints = int(map(716 ,width, 6, 60));

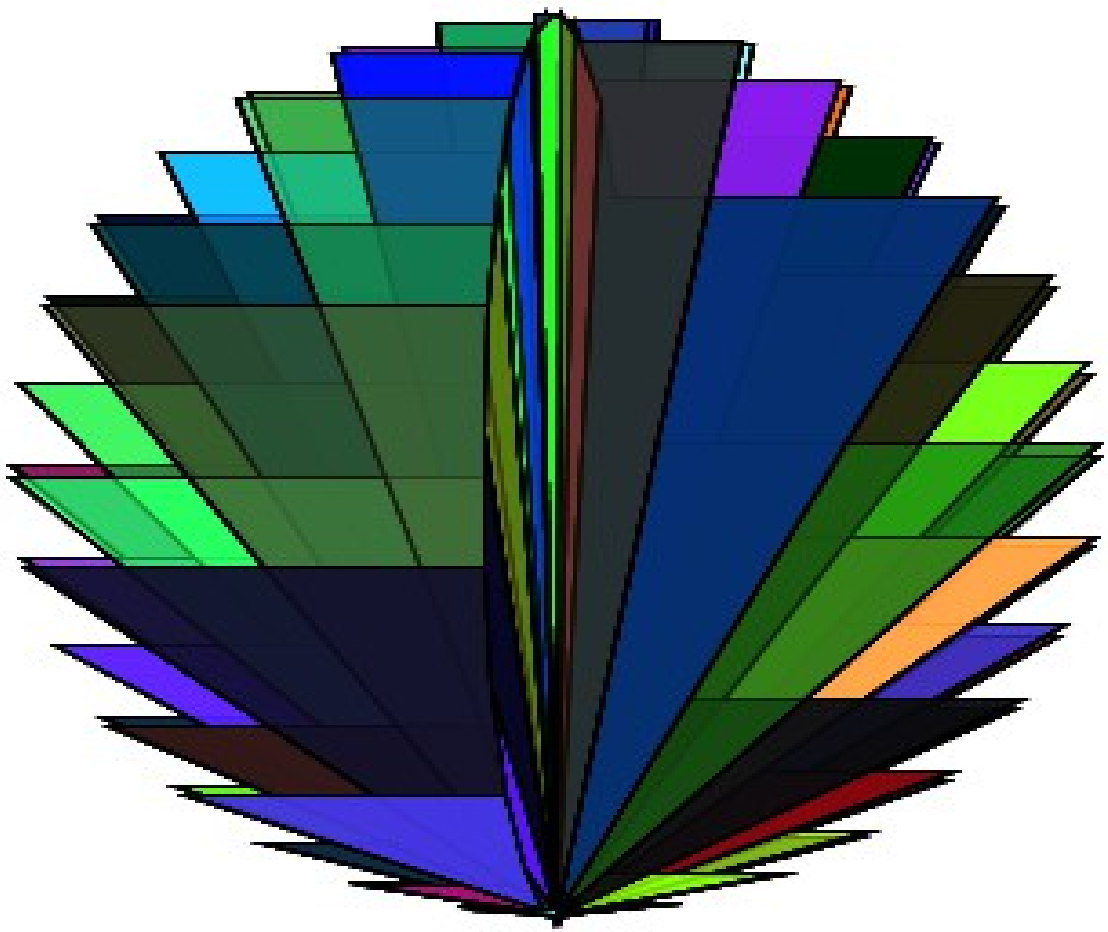


Illustration 98

```
int numPoints = int(map(294 ,width, 6, 60));
```

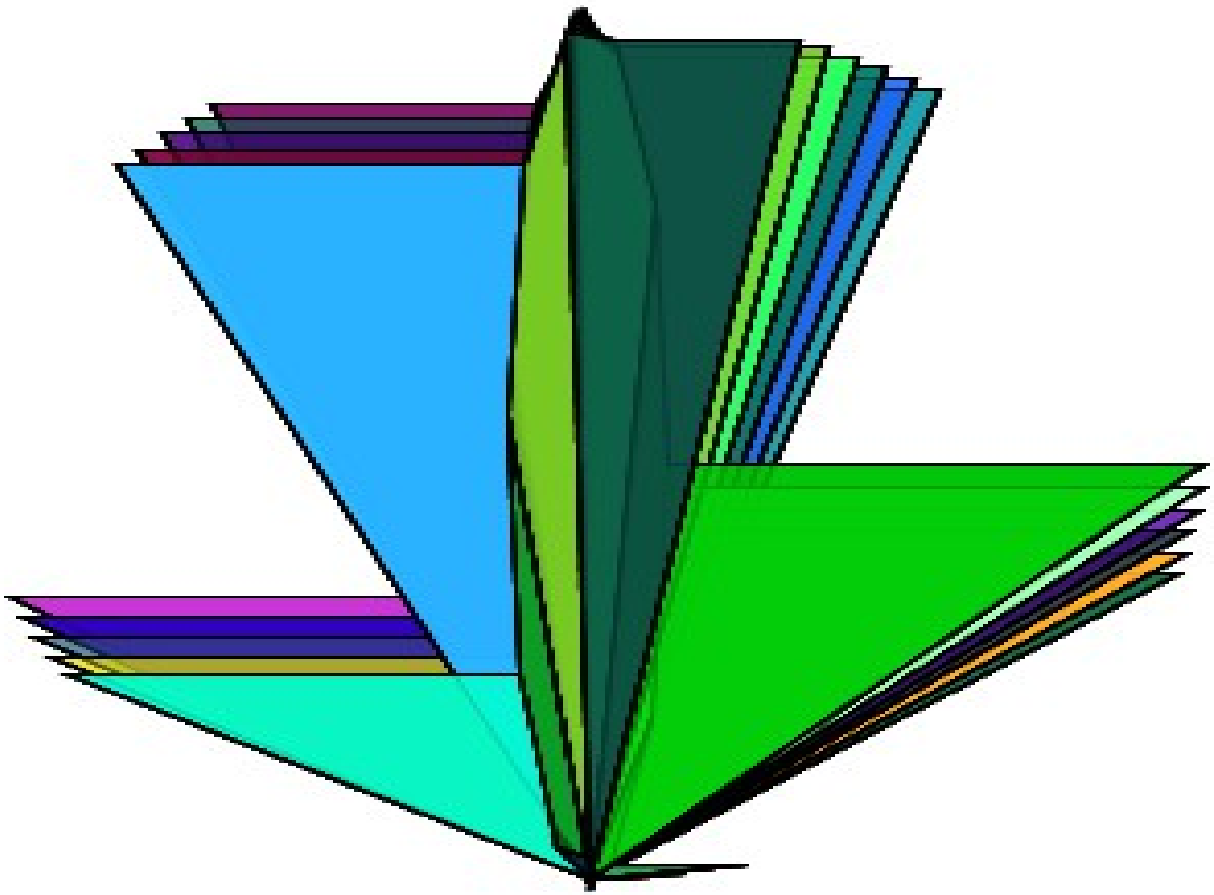


Illustration 99

```
int numPoints = int(map(523 ,width, 6, 60));
```

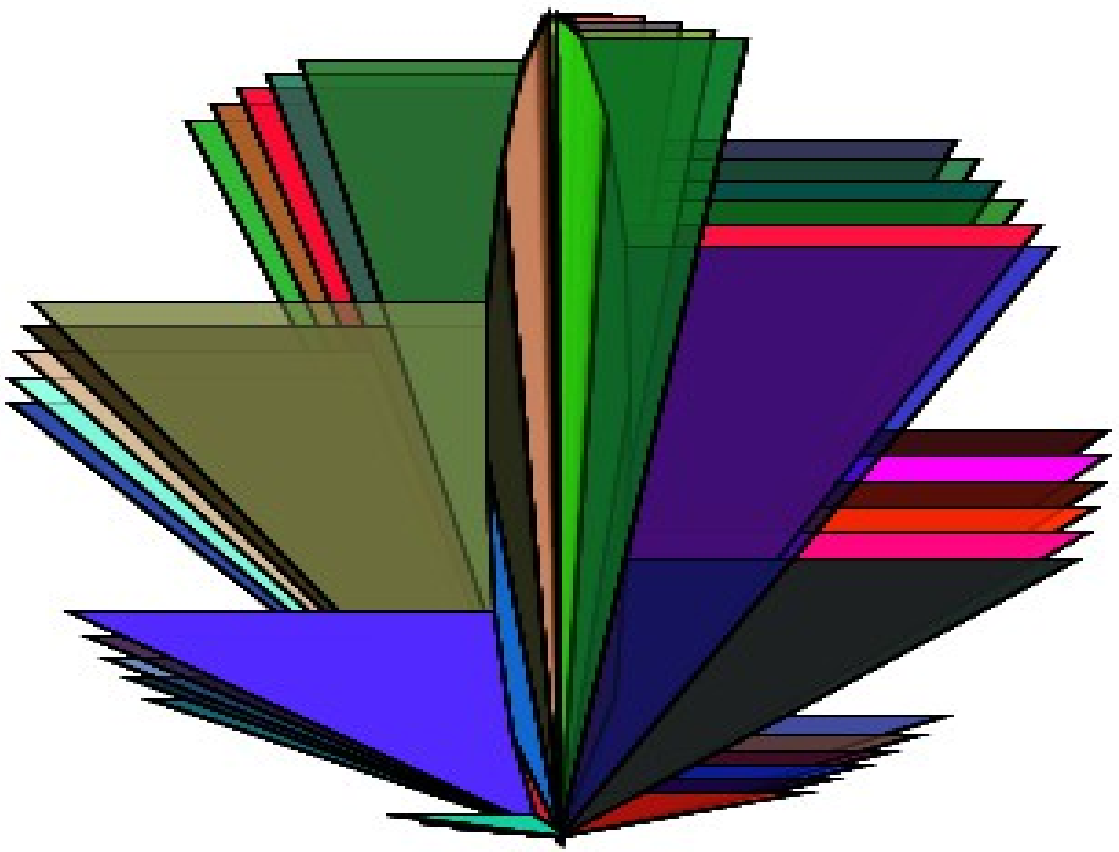


Illustration 100

```
int numPoints = int(map(596 ,width, 6, 60));
```

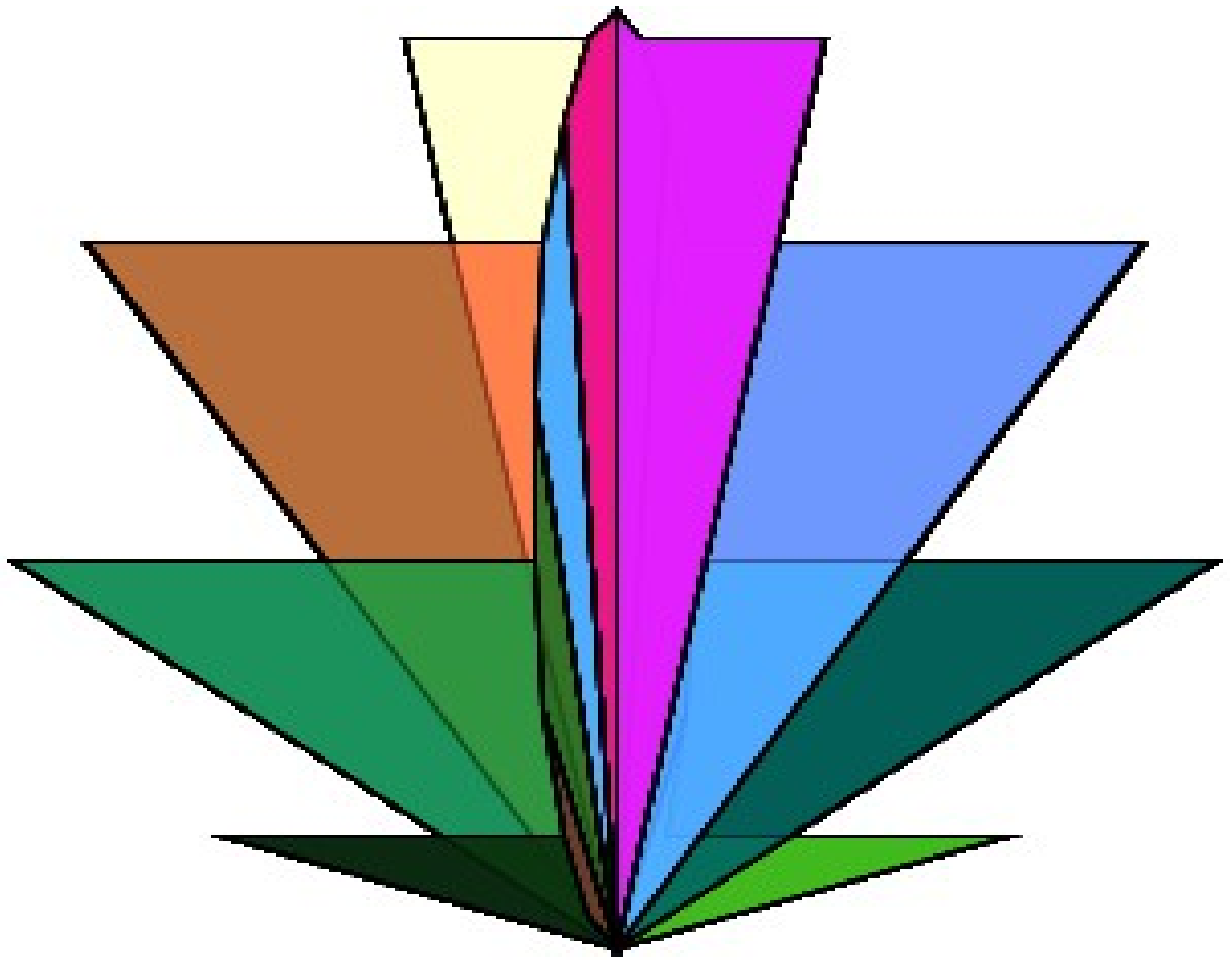


Illustration 101

Shape 31

```
beginShape(PShape.QUAD_STRIP);
for (int i = 0; i <= numPoints; i++) {
  float px = x + cos(radians(angle1-angle)) * outsideRadius;
  float py = y + sin(radians(angle1-angle)) * outsideRadius;
  angle += (angleStep);
  fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
  vertex(px, py);
  px = x + sin(radians(angle1-angle)) * insideRadius;
  py = y + sin(radians(angle1-angle)) * outsideRadius;
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
  angle1 += (angleStep);
  px = x + cos(radians(angle1)) * outsideRadius;
  py = y + sin(radians(angle1)) * insideRadius;
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
  angle += (angleStep);
  px = x + cos(radians(angle)) * insideRadius;
  py = y + cos(radians(angle)) * insideRadius;
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
  angle += (angleStep);
}
endShape();
```

```
int numPoints = int(map(487 ,width, 6, 60));
```

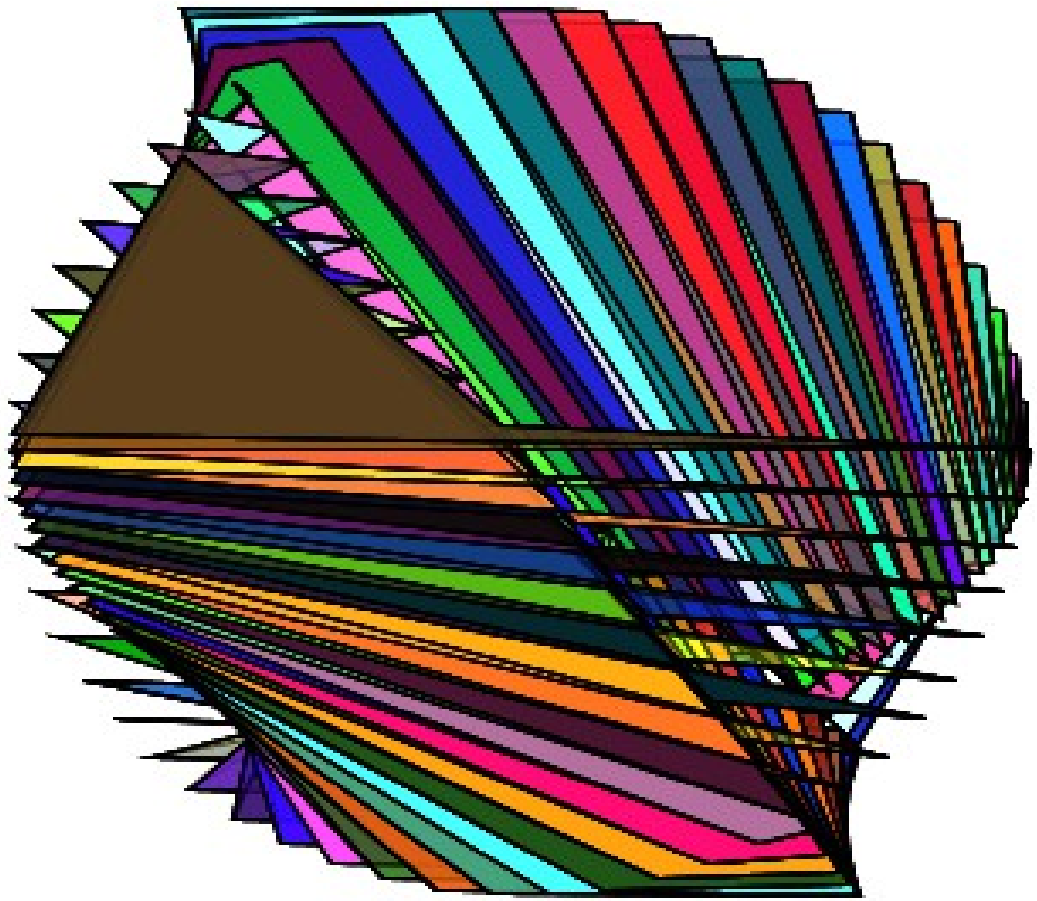


Illustration 102

```
int numPoints = int(map(456 ,width, 6, 60));
```

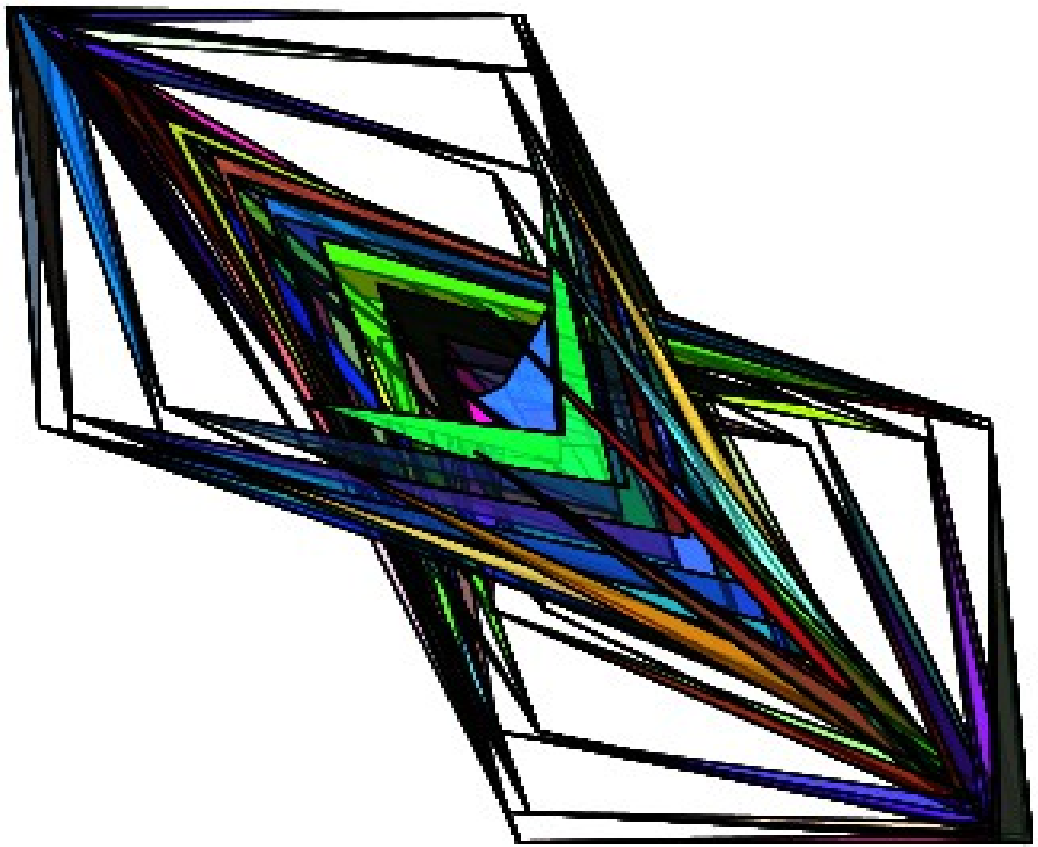


Illustration 103

```
int numPoints = int(map(62 ,width, 6, 60));
```

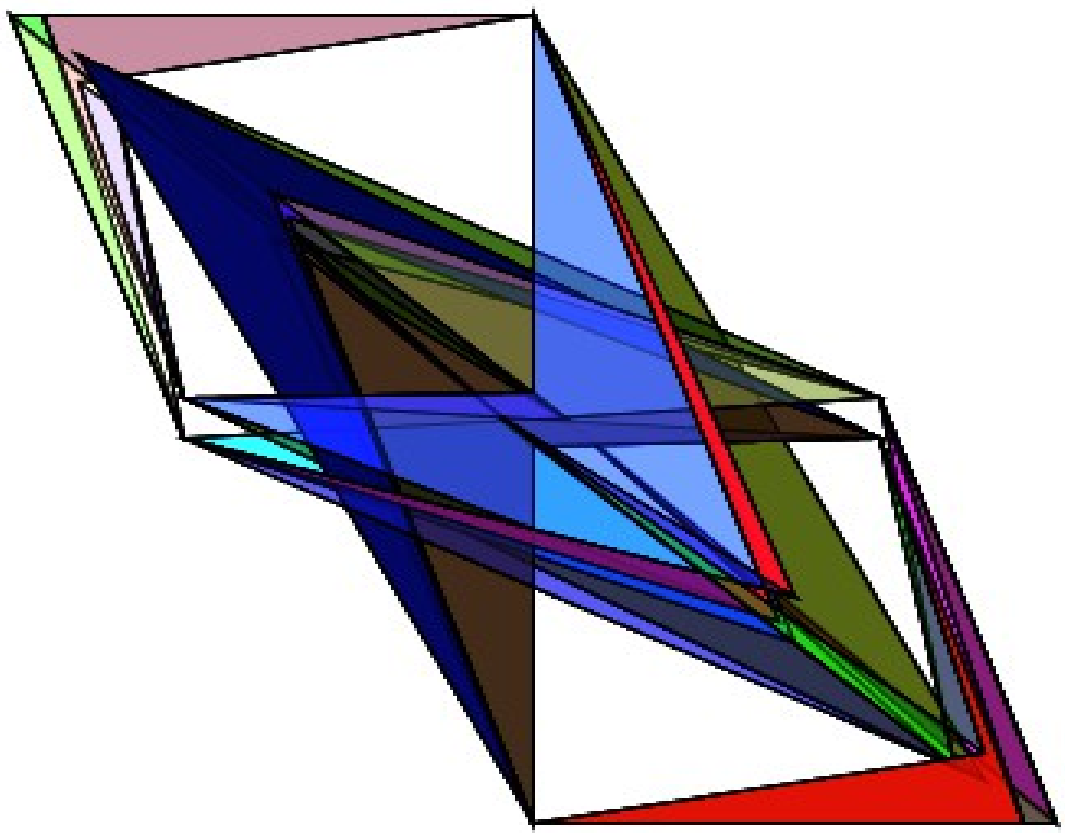


Illustration 104

Shape 32

```
beginShape(PShape.TRIANGLE_STRIP);
 for (int i = 0; i <= numPoints; i++) {
   float px = x + cos(radians(angle)) * insideRadius;
   float py = y + sin(radians(angle)) * insideRadius;
   angle += (angleStep);
   fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
   vertex(px, py);
   angle1 += (angleStep);
   px = x + cos(radians(angle1)) * insideRadius;
   py = y + sin(radians(angle1)) * outsideRadius;
   fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
   vertex(px, py);
   angle1 += (angleStep);
   px = x + cos(radians(angle1)) * insideRadius;
   py = y + sin(radians(angle1)) * outsideRadius;
   fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
   vertex(px, py);
   //angle += (2*angleStep);
 }
 endShape();
```

```
int numPoints = int(map(501, 0, width, 6, 60));
```

Illustration 105

Shape 33

```
beginShape(PShape.QUAD_STRIP);
 for (int i = 0; i <= numPoints; i++) {
  float px = x + cos(radians(angle)) * insideRadius;
  float py = y + sin(radians(angle)) * insideRadius;
  angle += (angleStep);
  fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
  vertex(px, py);
  angle1 += (angleStep);
  px = x + cos(radians(angle1)) * insideRadius;
  py = y + sin(radians(angle1)) * outsideRadius;
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
  angle1 += (angleStep);
  px = x + cos(radians(angle1)) * insideRadius;
  py = y + sin(radians(angle1)) * outsideRadius;
  fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
  vertex(px, py);
  //angle += (2*angleStep);
 }
 endShape();
```

```
int numPoints = int(map(430, 0, width, 6, 60));
```

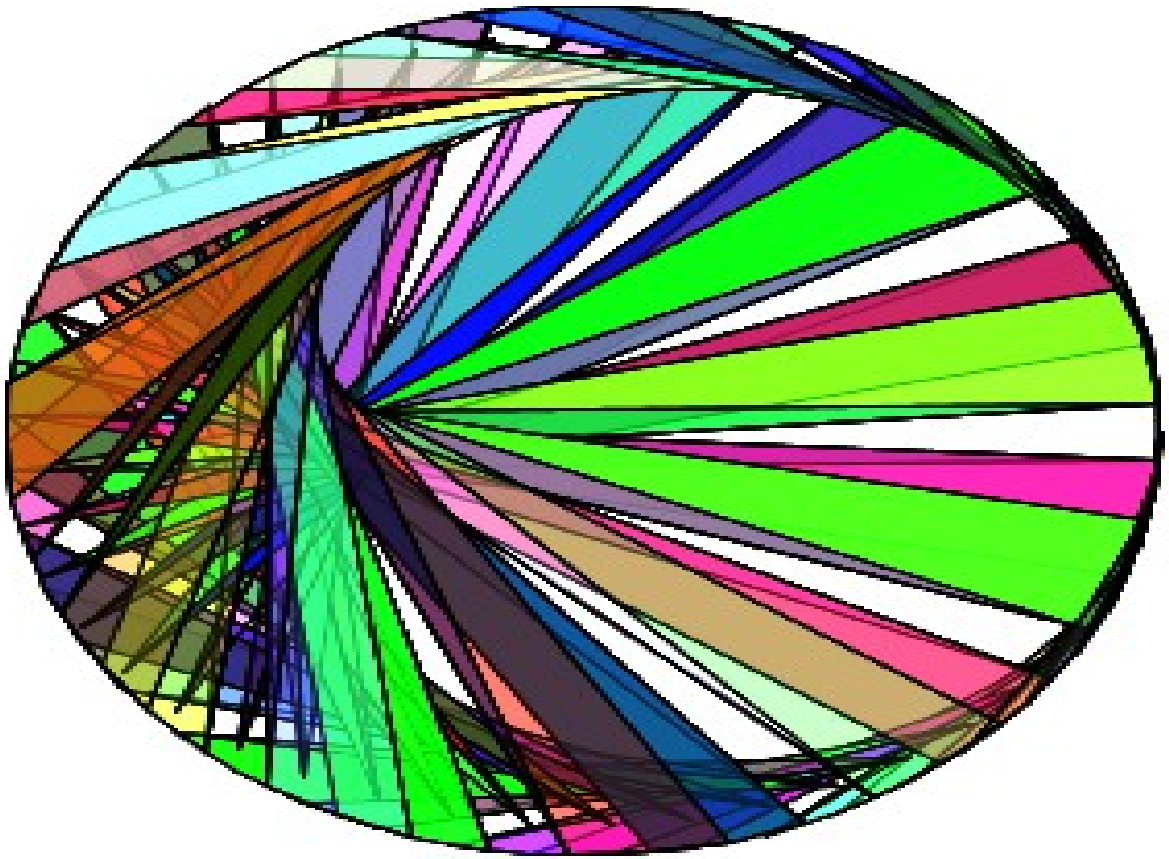


Illustration 106

```
 beginShape(PShape.TRIANGLE_STRIP);                                          Shape 34
 for (int i = 0; i <= numPoints; i++) {
   float px =x + sin(radians(angle)) * insideRadius;
   float py = y + cos(radians(angle)) * insideRadius;
   angle += (angleStep);
   fill(random(0, 245), random(10, 245), random(20, 255), random(30, 245));
   vertex(px, py);
   px = x+ sin(radians(angle)) * outsideRadius; ;
   py = y+ cos(radians(angle)) * outsideRadius;;
   angle += (angleStep);
   fill(random(40, 245), random(50, 255), random(60, 245), random(70, 255));
   vertex(px, py);
 }
 endShape();
}
```

```
int numPoints = int(map(241, 0, width, 6, 60));
```

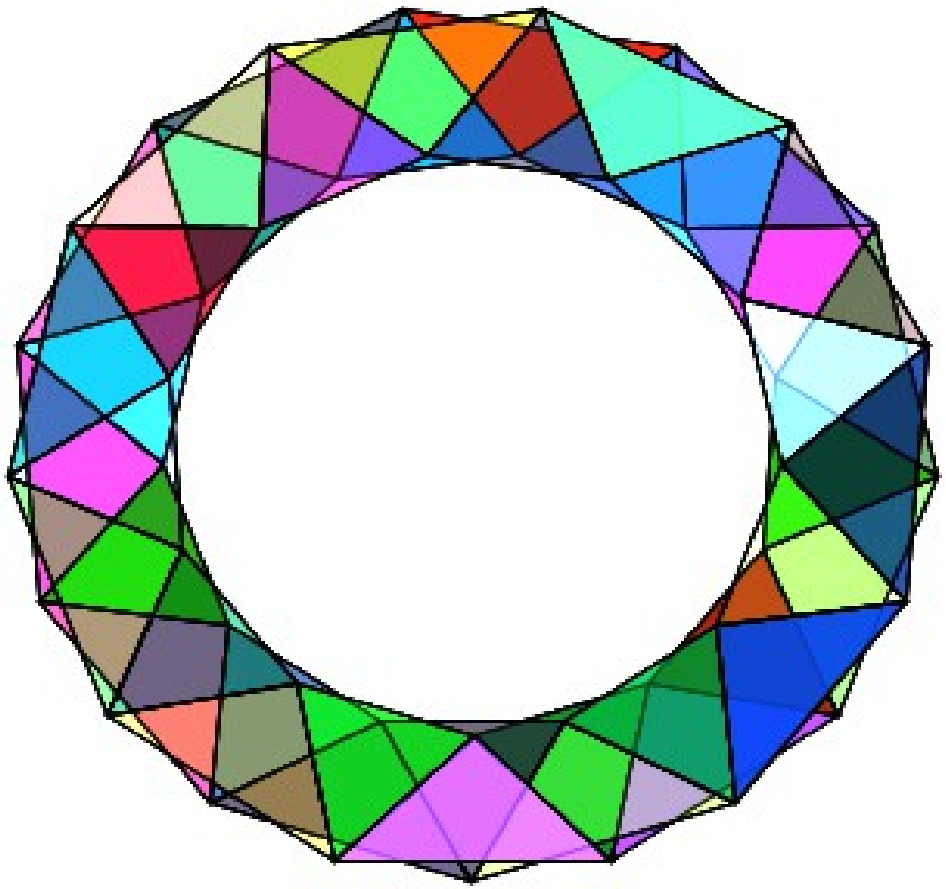


Illustration 107

```
int numPoints = int(map(160, 0, width, 6, 60));
```

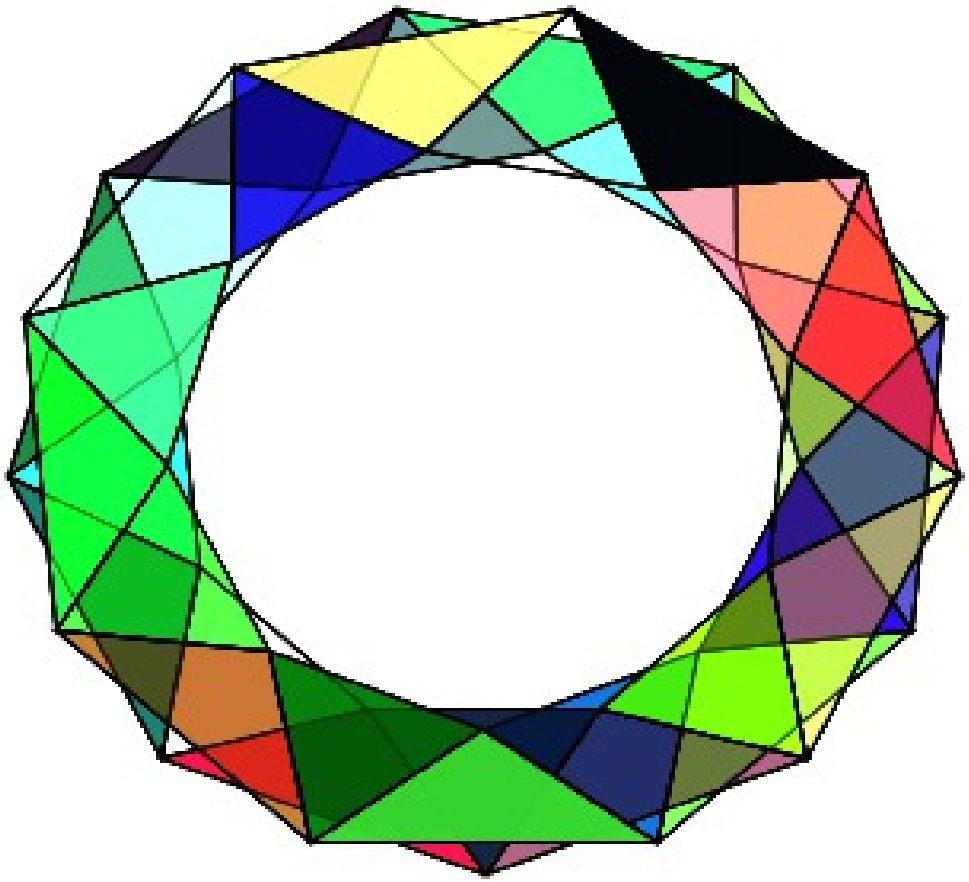


Illustration 108

```
int numPoints = int(map(650, 0, width, 6, 60));
```

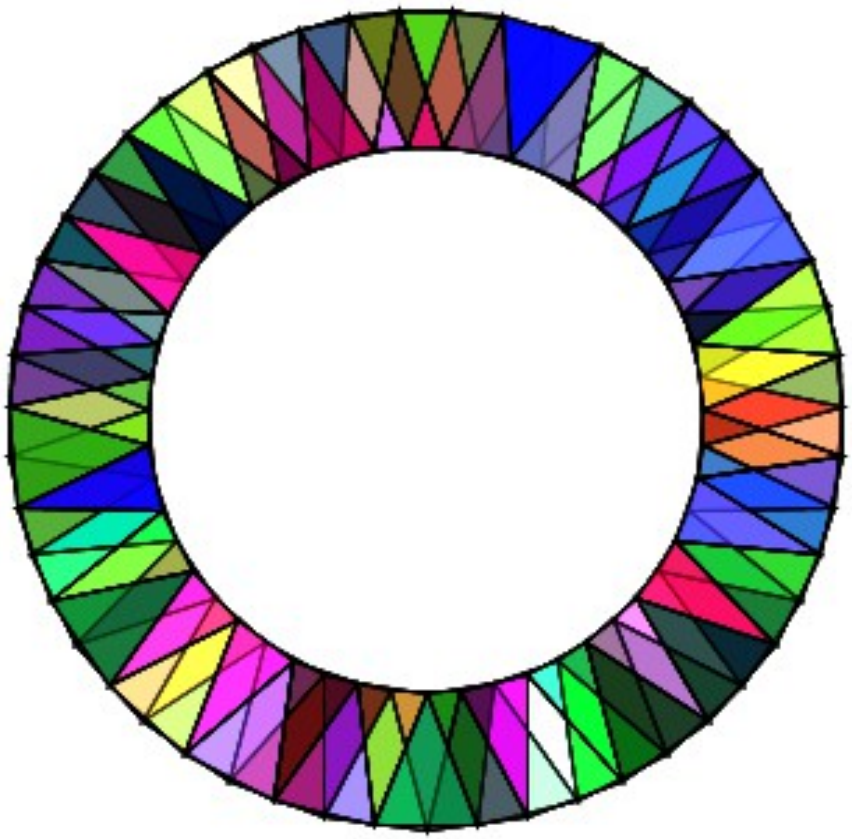


Illustration 109

APPENDIX A

CreativePlanet Source Code

CreativeExample Source Code

Alphabetical Index

A

Active.....30, Animation .....45, Art era 21

C.....

Aesthetics + Computation.....10, Art creation.....21, Audio communication 44

D.....

Application data.....46

H.....

Art history.....8

L.....

Art life.....5

P.....

Aesthetics part . 31, Application program 46

S.....

Amazing skills.....28

V.....

Analog voltages.....43

W.....

Application Window.....46

B

Background .....36, Beauty .....4, 6, 17, 28, BeginShape .....37, Belief .....6, Bit map .....44, BufferedImage 44

C.....

Blink Comparator.....53

blinkComparator().....47

C

Cartography.....42, Cartoons .....43, Characters .....43, Compression .....45, 52, Computer .....42, Copy-run-display .....61, Cosine .....6, Creation .....42, CreativeExamples .....48, CreativePlanet .....46, Creatives .....4, 6, 17, 28, CRT 43

D.....

Calligraphic displays..43, Color Division 30

E.....

coordinate endpoint.....43

L.....

Circuit layouts.....42

M.....

Computing machine.....21

P.....

Character plotting. .43, Computation part .....31, Creative process 19

S.....

Computer and Software.....6

T.....

Creative Triring.....38

D

B.....

Display buffer memory.....43

D.....

Digital…4, 6, 17, 21p., 25, 28, 33p., 36p.

Devices.....44, Display .....42, Display devices .....43, Draw .....35, Drawing .....46, DualEffect .....56, Dynamic 2D 45

E.....

Digital exercise.....4, Digital exercising .....28, Dual Effect 53

L.....

Display list.....43

O.....

Digital outputs.....4

P.....

Digital proves.....28, Display primitives .....43, Display processor 43

S.....

Display instructions.....46

dualEffect().....47

E.....

B.....

Electron beam.....43

C.....

Even Comparator.....53

D.....

Engineering Doctorate….5, Enjoy digital 28

EndShape.....35, Engineering

.....5, Enjoyment
.....6, Environment
.....6, EvenOrOddComparator
.....55, EvenOrOddComparator()
.....47
F
B.....
Floating billboard.....47
C..... H
Flash Comparator.....53
Facility.....31, Fascination
.....6, File system
.....46, Film
.....42, Finer
.....45, Fixed-animated I
.....47, FlashComparator
.....53, FlashComparator()
.....47, Flicker
.....43, FrameRate
.....35, Framework
.....46, Function-based
.....31
P.....
Formal proof.....28, flatbed plotters
.....42
R.....
Frame rate.....45 J
G
A..... K
Ghana Academy.....6
F.....
Graphics facility.....4 L
Games.....42, Ghana
.....6, Graphics
.....4
I.....
Graphic image.....44
P.....
Game processors.....42
S.....
Graphic shape.....6, Graphics system
.....46
Hardware.....43, Height
.....35, Home
.....42
I.....
Hands-free input.....44
C.....
Interactive Command.....47
Imaginary.....40, 42, Inch
.....45, Instrument
.....8
P.....
Interactive paradigm.....47
T.....
Input technology.....43
image(img)...47, text("FASCINATION ")
.....47
J2SE.....12
K
Keyboard .....46, Keyboards
.....42
L
C.....
Look Component 53
EvenOrOddComparator.....55, Library
.....31, Light
.....45, Lines
.....43, LinkComparator
.....54, Linux
.....12, Look
.....58
O.....
Light output.....43
P.....
Light pen.....44
M.....
D.....
Menu display.....46, Menu Displaying
.....46
Manipulation.....42, Map
.....6, 35, Mathematics
.....5p., 21p., MIT media lab
.....26, Modes
.....30, Mouse
.....6, menu()
.....47
O.....
C.....
Odd Comparator.....53
Object-oriented-based.....31, Octagon
.....36, OpenGL
.....4
T.....
Output technology.....43
P.....
C.....
Phosphor coating.....43, Pie charts
.....42, Projection

.....53, Proof creatives
.....28 R
D.....
Phosphor decays.....43
E.....
Picture elements.....43
F.....
Processing facility.....4
G.....
Playing games.....42
L.....
Program layout.....31, Programming
language.....10
Panel.....44, Paper
.....42, Pels S
.....43, Pictures
.....42, Pixels
.....43, Pleasure
.....4, Plotting
.....42, Primary principles
.....28, Processing
.....4, Program
.....31, Project
.....54, Project()
.....47, Proves
.....4
V.....
Persistence of vision.....45
W.....
Physical world.....45, Planet World
.....46
peom().....47, polygons
.....43
Q
Quadring.....37
B.....
Recreational beauty.....4, Refresh buffer
.....43
C.....
Recreational coding 4
F.....
Raster formats.....45
G.....
Raster graphics.....45
Real.....40, 42, Rectangle
.....30, Repeating
.....6
A.....
Simulation and Animation.....42
B.....
Skill builder.....6, Skill building
.....4
C.....
Section Comparator....53, Shape code art
.....61 T
F.....
Science fiction.....42
I.....
Static imaging.....46, Still images
.....45
M.....
Static mode.....30
P.....
Shape processing.....6
Science.....42, Screen
.....44, Section-screen
.....47, SectionComparator
.....57, Series
.....45, Setup
.....35, Sine
.....6, Size
.....35, Snippet
.....30, Software
.....43, 46, Solid
.....43, Solid-state memory
.....44, Something
.....6, Source
.....28, Static
.....30, Storage
.....42, Stroke
.....43, Synthesis
.....40, 42
T.....
Skill topic.....6, Storage tube display
.....42
W.....
Screen-based window.....46
sectionComparator().....47
T.....
A.....
Text Animation.....53
G.....
Text Graphics.....46
O.....
Text objects.....45
Tablet.....44, Techniques
.....43, Technology
.....42, Theory
.....28, Three-dimensional
.....42, Trigonometric

.....6, Two-dimensional
.....42, text("FASCINATION ")
.....47
U
C.....
User-computer creation.....43
V
C.....
Vertex coordination.....6
G.....
Vector graphics.....45
I.....
Vector information.....45
R.....
Variable resolution....45, Video Recorder
.....42
Vector.....43, Vector-based
.....45
W.....
Width.....35